SEL-71-007

Parallel Implementation of a Single Assignment Language

by

D. D. Chamberlin

January 1971

Technical Report No. 13

This work was supported in part by the Joint Services Electronics Program U.S. Army, U.S. Navy, and U.S. Air Force under contract N-00014-67-A-0112-0044 and by the National Aeronautics and Space Administration under Grant 05-020-337.

DIGITAL SYSTEMS LABORATORY

STANFORD ELECTRONICS LABORATORIES

STANFORD UNIVERSITY • STANFORD, CALIFORNIA

SEL-71-007

PARALLEL IMPLEMENTATION OF A SINGLE ASSIGNMENT LANGUAGE

by

D. D. Chamberlin

January 1971

Technical Report no. 13

DIGITAL SYSTEMS LABORATORY

Stanford Electronics Laboratories

Stanford University

Stanford, California

This work was supported in part by the Joint Services Electronics Program U.S. Army, U.S. Navy, and U.S. Air Force under contract N-00014-67-A-0112-0044 and by the National Aeronautics and Space Administration under Grant 05-020-337.

PARALLEL IMPLEMENTATION
OF A SINGLE-ASSIGNMENT LANGUAGE

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING

AND THE COMMITTEE ON THE GRADUATE DIVISION

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

By

Donald Dean Chamberlin

January 1971

ABSTRACT

This thesis describes a high-level computer programming language, called SAMPLE, and a parallel processing system to implement the language. SAMPLE belongs to the class of single-assignment languages, which have the property that statements are not necessarily executed in their order of appearance in the program; rather, each statement is triggered by the readiness of the data on which it depends. Because of this property, single-assignment languages are well adapted for parallel processing.

Rules are given for compiling SAMPLE programs into machine-level instructions, and a machine organization is described to execute the resulting code. During execution of a program, many processors are active simultaneously, each with its own independent instruction stream. Expandability and graceful degradation are intrinsic properties of the system organization.

Some experiments are described which simulate the behavior of the proposed system and compare it with a conventional, single-processor system. It is concluded that the proposed system offers a speed advantage over a conventional system, at the expense of increased processor costs and memory requirements.

ACKNOWLEDGEMENTS

I wish to thank my major adviser, Professor E. J. McCluskey, for his constant support and guidance in the work reported here. Thanks are also due to Professors A. M. Peterson and L. A. Manning for their help in preparing the document.

I owe a special debt of gratitude to Professors W. M. McKeeman, of the University of California at Santa Cruz, and H. S. Stone of Stanford University, whose generous flow of ideas provided much of the direction of my work, and to my wife, Judy, whose help and encouragement made it all possible.

This work was conducted under the support of a National Science Foundation Graduate Fellowship, with additional support provided by the Joint Services Electronics Program, and by National Aeronautics and Space Administration Grant No. NGR-05-020-337. Editing and preparation of the text was done using WYLBUR, the terminal-based text-editing service of the Stanford Computation Center.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE

INTRODUCTION

DEFINITION OF THE PROBLEM

This thesis describes a means of organizing a digital computer system for parallel processing. For our purposes, parallel processing is defined as the simultaneous use of more than one processing unit in processing a computer program. Depending on the details of its organization, a parallel computing system may have any of the following advantages:

1. Its net speed in processing a given program may be faster than that of a system with only one processor.
2. Its organization may be more flexible than that of a single-processor system. For example, more processors, and hence more computing power, may be added without changing the basic organization of the system.
3. In the event of a failure, other components may be able to take over the function of the failed component, resulting in a "graceful degradation" of performance.

4. The duplication of identical parts may afford some economies of scale in construction of the system.

This thesis will describe a system organization capable of realizing all of the above advantages.

In order to focus the computing power of multiple processors on a single program, some means must be found of breaking up the program into units of work, and assigning the units of work to the processors. Ideally, all the processors would be active at all times without mutual interference. In practice, the parallelism achievable in processing any given algorithm is limited by the nature of the algorithm. Any well-defined algorithm can be thought of as a collection of units of work, with certain sequential dependencies among the units of work which define the order in which the units must be completed, in order to arrive at the correct result. If the algorithm contains within it some set of units of work, none of which depends in any way on the completion of the others, it may be possible for these units of work to be executed simultaneously on parallel processors. The fundamental problem of parallel processing is the discovery of the sequential dependencies among the units of work in a given algorithm, and the allocation of processors to those units of work which can proceed in parallel.

HISTORICAL APPROACHES

The problem of parallel processing has been approached in a number of different ways. One way in which various approaches can be distinguished is the "level" at which one attempts to discover parallelism. Some workers have conceived the basic unit of work as a large portion of a program, such as a block or procedure. This has led to such language facilities as the multitasking feature of PL/I (24), in which more than one "task", each containing many statements, may be simultaneously active, but no attempt is made to find parallelism within a task. On a somewhat lower level, the basic unit of work might be considered a single statement in a high-level language. Thus, high-level language facilities have been proposed which permit creation of parallel flows of control among the statements of a program, such as Anderson's FORK and JOIN statements (3) and Opler's DO TOGETHER statement (31). On a still lower level, one might seek parallel units of work within a single expression. For example, in evaluating the expression (A+B)*(C+D), the two additions could be treated as parallel processes. Hellerman (20) and Stone (34), among others, have proposed algorithms for scanning such expressions and locating parallel work units.

Another way of distinguishing among approaches to parallel processing is by considering the machine on which the processing is to be done. Some machines, such as

SOLOMON (33) and ILLIAC IV (5), have a single instruction stream shared among many processors, all of which are constrained to execute the same instruction at the same time. Others, such as the CDC 7600 (12), contain multiple processors which are capable of acting independently and asynchronously. Still others, such as the IBM System 360 Model 91 (2), have a single instruction stream but introduce parallelism into its processing by means of lookahead and pipelining. A few radical, cellular organizations have been proposed, such as the Holland Machine (21) or Crane and Githens' Distributed Logic Memory (14); in these organizations, both computational and memory functions are distributed throughout an array of identical cells.

A third distinguishing feature among methods of parallel processing is the means by which sequential dependencies are discovered in the algorithm to be processed. One school of thought holds that the programmer should explicitly identify the units of work and their sequential dependencies; this has led, for example, to the FORK and JOIN statements and to the PL/I multitasking features described above. Another school of thought holds that a program in a conventional language should be analyzed automatically by the system in order to discover potential parallelism; work has been done on such analysis of Algol at the Burroughs Corporation (7), and similar work has been done on the FORTRAN language at UCLA (4). Other

workers have proposed entirely new languages for the expression of parallel algorithms, such as APL (26).

A fourth characteristic of parallel processing studies is their orientation toward theory or toward implementation. Some of the approaches described above, such as ILLIAC IV, have been intended as practical systems, and have actually been implemented. Other workers, such as Adams (1) or Karp and Miller (27), have proposed more abstract models of parallel processing, and have been able to construct theoretical proofs that their models possess certain properties, such as universality and determinacy.

A NEW APPROACH

In a paper presented at the 1968 Spring Joint Computer Conference (35), Larry Tesler and Horace Enea proposed a new class of programming languages called Single-Assignment Languages. In a single-assignment language, statements do not necessarily execute in the order in whcih they appear in a program; rather, each statement executes as soon as all the variables it needs are defined. In order that each statement be triggered at a well-defined time, it is required that each variable be assigned a value only once during the execution of a program. In a single-assignment program, there is no "flow of control" in the conventional sense; rather, the sequencing of statements is determined by the data flow, as some statements assign values to

variables which are needed by other statements. Single-assignment languages are well-adapted to parallel processing because, if many statements simultaneously have all their needed variables defined, they may all be executed in parallel.

This thesis describes a high-level single-assignment programming language, called SAMPLE, based on some of the ideas of Tesler and Enea. With respect to the four distinguishing characteristics described under "Historical Approaches", the present work might be categorized as follows:

1. The basic unit of work will be sought on as low a level as possible, even within a statement in the language. The decision to seek low-level parallelism was made in order to exploit every possibility for parallel processing, with the realization that this approach would probably be expensive in terms of the system overhead necessary to discover the potential parallelism.
2. The processing is to be done on a machine having many independent, asynchronous processors. Since the system consists of a variable number of processors, with no central control unit, it automatically possesses the properties of expandability and graceful degradation.
3. The programmer need not explicitly state the opportunities for parallelism in his algorithm;

they will be automatically discovered by the system. A programming language will be described having a structure such that the opportunities for parallel processing can readily be uncovered by the system.

4. The work described here is implementation-oriented. Several examples will be given of programs in the proposed language. Rules will be stated for compilation of the language into machine-level instructions, and a detailed description will be given of a hardware system, able to be built with presently-available components, to execute the resulting object program. Some results of a simulation of the proposed system will be described.

CHAPTER TWO

THE LANGUAGE

INTRODUCTION

Single-assignment languages, as defined by Tesler (35), are languages which require that each variable be assigned a value only once in a program. Each statement in a single-assignment language has certain input variables (on whose value the statement depends) and certain output variables (whose values are defined by the statement). As soon as all the input variables of a given statement are defined, that statement may be executed with assurance that none of its input values will ever change. Thus the order of execution of statements in a single-assignment language is implicitly determined by the data flow of the program, and the physical ordering of the statements is immaterial. Single-assignment languages are well-adapted to parallel processing because, at any point in time, a simple algorithm can determine the set of statements which are ready for execution.

Single Assignment Mathematical Programming Language, SAMPLE, has been developed for implementation on a parallel processing system. SAMPLE is intended for numeric programs having a high degree of implicit parallelism. It

provides a full range of arithmetic and logical operators, arrays, block structure, recursive procedures, conditional expressions, and iteration. It does not provide facilities for handling character strings or linked lists; however, these facilities could be added without altering the basic nature of the language.

EXAMPLE OF SINGLE-ASSIGNMENT PROCESSING

Before proceeding with a detailed description of the features of SAMPLE, an example will be given of the concept of single-assignment processing. Suppose that we are given two numbers A and B, and we wish to find which is greater, (A+B)/(A-B) or (A*B)/(A-B) . We might write the following program:

```
BEGIN
    W ← A-B;
    X ← (A+B)/W;
    Y ← (A*B)/W;
    Z ← X>Y;
END
```

At the conclusion of processing, the variable Z would contain a Boolean value of TRUE if (A+B)/(A-B) is greater than (A*B)/(A-B); otherwise it would contain the value FALSE.

We will now examine this program from the point of view of the single-assignment property. We notice that there are six operations to be performed. If we invent the names T1 and T2 for certain intermediate results, we could write the six operations as follows:

(1) W ← A-B

(2) T1 ← A+B

(3) X ← T1/W

(4) T2 ← A*B

(5) Y ← T2/W

(6) Z ← X>Y

Each operation has two inputs and one output. We will make the following assumptions:

A. Each operation requires one time-unit to complete.

B. Each operation is initiated as soon as its inputs are ready. If many operations are simultaneously ready, they proceed in parallel.

C. When we start processing, A and B are ready.

Then the sequence of events in processing the program is as follows:

1. In the first time-unit, operations 1, 2, and 4 have their inputs ready, so they are executed in parallel. This makes their respective outputs, W, T1, and T2, ready.

2. In the second time unit, operations 3 and 5

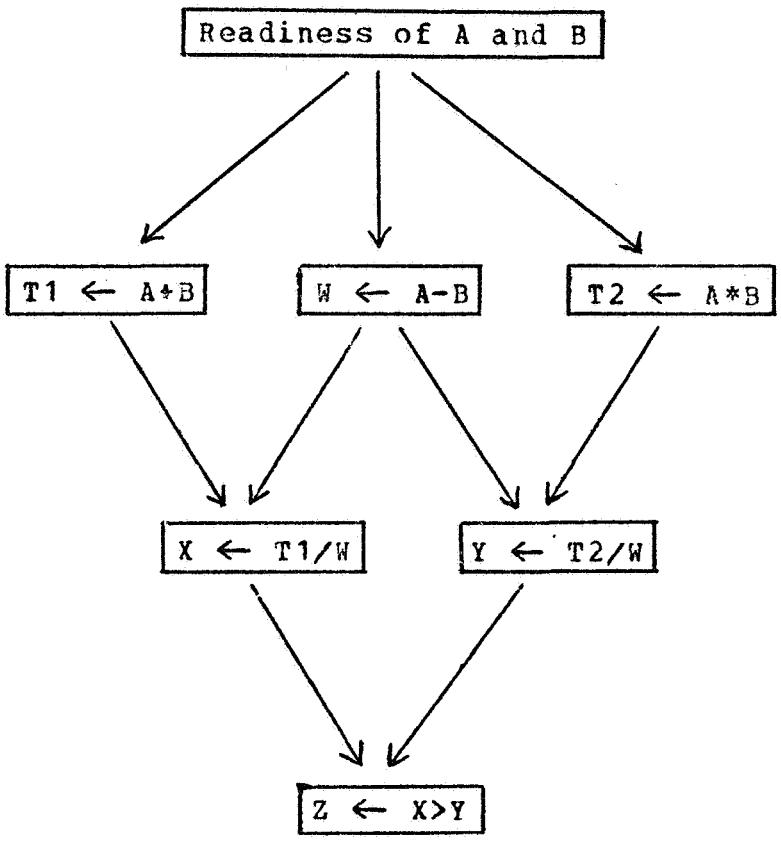


FIGURE 2.1

SEQUENTIAL DEPENDENCIES IN EXAMPLE PROGRAM

execute, making ready their outputs, X and Y.

3. In the third time-unit, operation 6 executes, making ready the final result, Z.

This sequence of events, showing the sequential dependencies among the six operations, is illustrated in Figure 2.1.

The example we have considered is a valid SAMPLE program, and the above steps reflect the way in which our proposed system might process the program. This same example will be considered in more detail in the chapters on Compilation and Execution.

DATA TYPES

SAMPLE has two data types: numbers and tuples. No distinction is made between integers and real numbers. Either a number or a tuple may be given a name, which consists of any string of alphabetic or numeric characters, the first of which must be a letter.

A constant number may be written in either integer or floating point form, with or without an explicit exponent. Examples: 2 -3.5 2.69E-8 -1E6

A tuple is an ordered set of elements, each of which may be a number or a tuple; thus multi-dimensional arrays may be tuples of tuples. Tuples are denoted by angle brackets, for example: <1, 2.5, X+Y> or <<1, 2>, <3, 4>> . The reserved words TO and BY may be used in writing tuples:

<A TO B BY D> means <A, A+D, A+2D, ... , B>

<A TO B> means <A, A+1, A+2, ... , B>

An individual element of a tuple may be referred to by means of the subscript operator ↓ . A ↓ I is the name of element I of tuple A. Tuple elements are usually numbered starting with 0; however, this rule can be superceded, as will be explained later. The built-in function FIRST A returns the subscript number of the first element of the tuple A; LAST A returns the subscript number of the last element of A. Successive occurrences of ↓ are grouped from left to right unless modified by parentheses, so A ↓ I ↓ J means (A ↓ I) ↓ J rather than A ↓ (I ↓ J).

ASSIGNMENT STATEMENTS

The assignment operator is the left arrow ← . The value of the expression appearing on the right of the arrow is assigned to the variable or tuple element appearing on the left of the arrow. Example: X ← Y ; assigns to X the value of Y.

Before values can be assigned to the elements of a tuple, upper and lower bounds must be defined for the subscripts of the tuple, in order that memory space may be allocated for its elements. Two methods are provided for defining these bounds; the programmer may use whichever method he finds most convenient. The first method is by

means of "bounding" statements, using the reserved word IS. The following sequence of statements states that X is a tuple having subscripts from -10 to 10, X ↓ I is a tuple having subscripts from 0 to N, and X ↓ I ↓ J is assigned the value of A + B :

```
X IS TUPLE (-10,10) ;
X ↓ I IS TUPLE (0,N) ;
X ↓ I ↓ J ← A + B ;
```

The second method of defining bounds for subscripts is by means of the reserved word OF, which may be inserted in an assignment statement. For example,

```
X ↓ 5 OF (0,10) ← U;
```

states that the tuple X has subscripts ranging from 0 to 10 and that X ↓ 5 is assigned the value of U. If the lower subscript bound is 0, it may be omitted; so

```
X ↓ 5 OF 10 ← U;
```

has the same meaning as the above statement. If a name on the left side of an assignment statement has multiple subscripts, each subscript must have its own OF clause. Thus, the following statement has meaning equivalent to the three-statement example in the last paragraph:

X ↓ I OF (-10,10) ↓ J of N ← A + B;

The two methods of defining tuple bounds must never be mixed in a statement. If a tuple has a special "bounding" statement, then any tuples nested inside it must also have "bounding" statements, and no OF clause may appear in any statement which assigns values to the elements of these tuples. Conversely, if a statement assigning a tuple element has an OF clause, then:

a. The tuple must not have a "bounding" statement.

b. All statements which assign values to elements of the tuple must have OF clauses, and all the bounds stated in the OF clauses must agree. This enables whichever statement may execute first to define the bounds of the tuple and allocate space for it.

c. All tuples nested inside the given tuple must have their bounds defined by means of OF clauses.

A subscript or subscript bound may be any expression. However, the programmer must ensure that his program does not violate the single assignment property. For example, if A ↓ J and A ↓ K appear on the left side of two assignment statements, J and K must not evaluate to the same number.

EXPRESSIONS

Four categories of arithmetic and logical operators are provided for constructing expressions. The operators may be mixed in any desired way. The precedence among the operators is given in a later section, and may be modified at will by the use of parentheses.

Boolean operators (AND, OR, NOT) interpret any negative input as TRUE and any non-negative input as FALSE. They store the value TRUE as -1 and FALSE as 0. The reserved words TRUE and FALSE may be used to represent -1 and 0, respectively.

Category I. Binary operators: + - * / MOD AND OR

If both operands are numbers, the result is a number. If one operand is a number and the other is a tuple, the operation is performed between the number and each element of the tuple, and the result is a tuple. If both operands are tuples, the operation is performed element-by-element, and the result is a tuple.

X MOD Y is defined as the difference between X and the largest multiple of Y not greater than X. None of X, Y, nor X MOD Y need be an integer.

Category II. Number unary operators: - ROUND FLOOR CEIL ABS NOT

-	Arithmetic negation
ROUND	The nearest integer to the operand (.5 rounds up)
FLOOR	The largest integer not greater than the operand
CEIL	The smallest integer not less than the operand
ABS	Absolute value
NOT	Logical negation

If the operand is a number, the result is a number. If the operand is a tuple, the operation is performed on each element of the tuple. The result is a tuple.

Category III. Tuple unary operators: + * AND OR

The operand must be a tuple. The result is a number created by joining all elements of the operand tuple together by the corresponding binary operator. For example, + <1,2,3> = 6 .

Category IV. Relational operators: = ¬= < <= > >=

These operators yield the value -1 or 0 if the relation is true or false, respectively. If one operand is a number and the other is a tuple, ¬= is TRUE and all other relations are FALSE. If both operands are tuples, and the operator is not ¬=, the relation is TRUE only if the tuples have identical "first" and "last" values, and the relation holds between each pair of corresponding elements in the tuples. The relation ¬= is TRUE between two tuples whenever the relation = is not TRUE.

The relational operators may be used anywhere in an expression. For example, A ← X > Y+Z ; assigns the value TRUE (-1) to A if X is greater than Y+Z, and FALSE (0) otherwise.

CONDITIONAL EXPRESSIONS

Conditional expressions have the following form:

```
IF X THEN Y ELSE Z
```

where X, Y, and Z may be any expressions (except those containing the reserved word WITH). Conditional expressions may be nested, and they may be used wherever an expression may be used.

WITH EXPRESSIONS

It is often necessary to change an element of a tuple while preserving the other elements. This would violate the single-assignment property unless the altered tuple were given a new name. The special expression A WITH I ← U is equal to the tuple A with element I replaced by the value of U. This type of expression may be used wherever an expression is called for, and may be nested as required. Example:

```
B ← A WITH IF X=Y THEN I ELSE J ← U ;
```

INPUT AND OUTPUT STATEMENTS

The order of execution of statements in a SAMPLE program is unpredictable. Therefore we cannot use the conventional scheme of providing the input medium with an ordered set of input quantities, which the program will call for in order as they are needed. Instead, all input quantities must be simultaneously available, and each input statement must be able to call from the input medium the particular input quantity it needs. We assume that the input medium is associative; each input quantity may be a number or a tuple (possibly having other tuples nested inside it), and each input quantity is associated with a unique integer. The basic SAMPLE input statement, READ, specifies the variable to be read, and the integer tag associated with it in the I/O medium. The READ statement causes the input medium to be scanned for the input quantity associated with the given integer tag; this quantity is then assigned to the given variable. The variable to be read may be either a simple name or a tuple element. If it is a tuple element, either of the two methods of defining subscript bounds may be used. The following example assigns to X the quantity associated with the number 1 in the input medium:

```
READ (X,1);
```

Of course, since the input statements execute in an unpredictable order, the numbers associated with the input

quantities do not imply any ordering among them. The programmer may specify several input quantities in a single statement by repeating the parenthesized portion of the statement. Examples:

```
READ (A,1), (B,2), (C,3);
READ (A ↓ J, N), (A ↓ K, N + 1);
READ (A ↓ I OF (1,10) ↓ J OF 2, 10 * I + J);
```

The SAMPLE output statement, WRITE, operates analogously to the READ statement. There is assumed to exist an associative output medium into which output quantities may be placed and associated with unique integers. Any expression may appear as the quantity to be output in a WRITE statement. The first of the three examples below causes the value of X to be placed in the output medium and associated with the number 1:

```
WRITE (X,1);
WRITE (A ↓ J, N), (A ↓ K, N + 1);
WRITE (A ↓ J * B ↓ K + C, 2 * J + K);
```

The associative input and output media might be discs or drums. A special I/O processor might be provided to convert information from the I/O media to a format more convenient for external use. Neither of these ideas is without precedent. The associative search for an input

quantity is similar to the "search key" commands implemented on IBM 2314 disc drives.(22) The idea of a special I/O processor to interface a parallel system with the external world is reminiscent of the ILLIAC IV project, which uses a Burroughs B6500 as an I/O processor.(5)

BLOCK STRUCTURE

In order that a programmer may write a section of code for insertion into a program without fear of duplicating a variable or function name used in the outer program, block structure is provided in the SAMPLE language. A block is any sequence of statements enclosed between the words BEGIN and END and beginning with a declaration of all variable or function names to be used in the block with a local meaning. Blocks may be nested inside each other to arbitrary depth. If any name declared in an inner block duplicates a name declared in an outer block, it behaves throughout the inner block as though it were a different name, unrelated to the duplicate name appearing in the outer block. Statements in an inner block may refer to global names--names declared in an outer block but not in the inner block. Every name used in a block must be declared in that block or an outer block. The outermost block is the program itself, which must be enclosed between BEGIN and END and followed by a period (.).

A function name must be declared in the block in which the function is defined. Details of function declarations

will be explained in the section entitled "FUNCTIONS".

Block structure in SAMPLE is intended only to ensure uniqueness of names. It does not, as in some languages, imply memory allocation upon block entry. Because SAMPLE statements are not executed in the order in which they are written, "block entry" has no meaning in SAMPLE. The data type of a name is not specified in a declaration. The following example program contains one nested block; it reads two quantities and writes their sum:

```
BEGIN
    DECLARE A, B, C;
    READ (A,1), (B,2);
    C ← A + B;
    BEGIN
        DECLARE A;
        A ← C;
        WRITE (A,1);
    END
END.
```

FUNCTIONS

A SAMPLE function behaves like an ALGOL procedure, with the restrictions that all its parameters must be called by value, and it must always return a single value and have no side effects. Each function is defined in the following way:

```
DEFINE <name> ( <name list> ) ; <function body> END
```

The <name list> is a list of the formal parameters of the function; i.e., those input quantities which must be supplied by the calling program each time the function is called. The <function body> consists of an optional declaration of variable names which are local to the function, zero or more statements to compute the value of the function, and, finally, an expression equal to the value of the function. The following are two equivalent definitions of a function which returns the arithmetic mean of its two parameters:

```
DEFINE MEAN(X,Y); (X+Y)/2 END
```

```
DEFINE MEAN(X,Y);
    DECLARE W, Z;
    W ← X+Y;
    Z ← W/2;
    Z
END
```

A function has many of the properties of a block. It may declare local names, and its parameter names are implicitly considered to be declared as local names by appearance in the parameter list. Function definitions must always occur in the head of a block, just after the

declaration for that block. A function name may duplicate another function name defined in another block; the name will then refer to different functions in the different blocks. Statements inside a function definition may refer to global names; i.e., names which are not declared in the function. However, a statement inside a function may not assign a value to a global name. The single-assignment property must be obeyed inside function definitions. A function definition may contain any type of statement except another function definition.

A function name must be declared in the block in which it is defined. The function names which are defined in a block are simply inserted in the declaration for that block, following the reserved word FUNCTION. A function call consists of the function name followed by a list of parameters in parentheses, and may be used in any expression. The number of actual parameters in a function call must equal the number of dummy parameters in the function definition. A function may call itself recursively.

In the following example block, functions SQUARE and MEANSQR are defined and used to find the mean of the squares of two numbers:

```
BEGIN
    DECLARE A, B, C, FUNCTION MEANSQR, SQUARE;
    DEFINE MEANSQR(X, Y);
        (SQUARE(X) + SQUARE(Y)) / 2
    END
    DEFINE SQUARE(W);  W * W  END
    READ (A,1), (B,2);
    C ← MEANSQR(A,B);
    WRITE (C,1);
END
```

ITERATION

Iteration is denoted by using, anywhere in a statement, a name enclosed in single quote marks. This indicates that the quoted variable is a tuple, and that the statement is to be executed once with each element of the tuple substituted for the quoted name. Only a name, and not an entire expression, may be quoted. If the same quoted name appears more than once in a statement, all copies of the quoted name are replaced by the same tuple element in each iteration. If many different quoted names appear in a statement, the statement will be repeated once for every possible different way of substituting a tuple element for each quoted name. The following example assigns to A the transpose of an N x N matrix B:

```
I ← <1 TO N>;
J ← <1 TO N>;
A IS TUPLE(1,N);
A ↓'I'IS TUPLE(1,N);
A ↓ 'I' ↓ 'J' <- B ↓ 'J' ↓ 'I';
```

The appearance of an iterated statement in a program has exactly the same effect as the appearance of a different statement for each iteration. The order of execution of the iterations is determined by the readiness of their respective input values. If many of the iterations are ready at the same time, they may be executed simultaneously.

LOOPS

The above convention of iteration by quoted tuple names is intended to give the programmer maximum possible parallelism, in the sense that each iteration is released for execution as soon as all its input variables are ready. This implies that some system resources (at least some memory space) must be allocated to each iteration of the statement at all times, until it has been executed. However, in some processes it is known in advance that the (I+1)th iteration depends on the results of the Ith iteration, and so it would be foolish to allocate system resources to all the iterations at once. Furthermore, some

algorithms, such as iteration until a condition is satisfied, are difficult or impossible to program using the quoted tuple convention. Therefore, two additional constructs have been provided: the FOR loop and the WHILE loop. The formats are as follows, where <expr> represents any expression not containing the reserved word WITH:

```
FOR <name> <- <expr> STEP <expr> UNTIL <expr> DO
     ┌──────────────────────┐
     │                      │
     │    <initialization>  │
     │      <loop body>     │
     │                      │
     └──────────────────────┘
END
```

```
WHILE <expr> DO
     ┌──────────────────────┐
     │                      │
     │    <initialization>  │
     │      <loop body>     │
     │                      │
     └──────────────────────┘
END
```

Loops obey the following rules:

1. The loop body is a set of statements or nested loops. The entire loop body is executed repeatedly, and each iteration is not begun until the previous iteration has finished. The FOR loop executes its body once for every indicated value of its index variable. The WHILE loop continues iterating as long as its condition is satisfied.
2. The loop body must obey the single-assignment property, both within itself and with respect to the

external program. The order of execution of statements inside the loop is governed by their data dependencies.

3. For each variable which is defined in the loop, at any point in time, two values are retained: the value computed in the present iteration, and the value computed in the previous iteration. The term OLD X refers to the value of X computed in the previous iteration. The operator OLD may be used in any expression inside a loop; however, its operand must be a variable defined in the loop or in an outer loop (not in an inner loop).
4. During the first iteration of the loop, the OLD values of the variables are defined by the initialization. Every variable appearing with the operator OLD must also appear in the initialization. The following example shows the format of an initialization:

```
INITIAL A ← 1, B ← X ↓ I, C ← <0 TO N>;
```

5. The index variable of a FOR loop must be declared, like any other variable in the program.
6. The continuation condition of a WHILE loop may not contain any OLD references, nor any variables defined in the loop but not initialized. When a WHILE loop is executed, variables appearing in the

initialization are set to their INITIAL values. The continuation condition is then tested; if it fails, no iterations occur, and the variables retain their INITIAL values. If the continuation condition succeeds, the initial values of the variables become the OLD values, and the first iteration proceeds. After each iteration, the newly computed values are used to determine whether another iteration should occur.

7. Global variables (defined outside the loop) may be used within the loop. These variables have the same value in every iteration.
8. If a statement outside a loop uses as an input quantity a variable defined in the loop, that statement must wait until all iterations of the loop are complete. Then, when the index variable has been exhausted or the continuation condition has failed, all variables defined in the loop are considered to be ready, and their values are the values computed in the last loop iteration. Note that this implies "levels" of readiness; a variable defined in a loop may be "ready" to other statements in its loop, because it has been defined for a particular iteration, but "not ready" to statements outside the loop, because all iterations are not yet complete.

9. The two types of loops may be nested inside each other without restriction up to eight levels of nesting. Before any iteration of an outer loop is considered complete, all iterations of all inner loops must be complete.
10. Statements iterated by the quoted tuple name convention may appear in a loop body.
11. No statement outside a FOR loop may use the index variable of the loop as an input variable.
12. The STEP clause of a FOR loop may be omitted, in which case the step is taken to be 1.
13. No function definition may occur inside a loop; however, a loop may occur inside a function definition, and a function call may occur inside a loop.

The following example of a loop computes N factorial:

```
FOR I ← 1 UNTIL N DO
    INITIAL X ← 1;
    X ← OLD X * I;
END
```

To illustrate nested loops, we might nest the above loop inside another loop. The program below computes N, the smallest integer whose factorial is greater than 100:

```
WHILE FACT<100 DO
    INITIAL N ← 1, FACT ← 1;
    N ← OLD N + 1;
    FACT ← X;
    FOR I ← 1 UNTIL N DO
        INITIAL X ← 1;
        X ← OLD X * I;
    END
END
```

SPECIAL VALUES

The programmer may use the special value NIL in expressions. NIL is the status of a name before its value is assigned. Any expression containing the word NIL will never become ready, and assigning such an expression to a name is the same as no assignment. Therefore, a given name may appear on the left side of several assignment statements without violating the single-assignment property, provided that at most one of the values assigned to the name is not NIL. The following example assigns to X the subscript number of the element in the tuple A which is equal to Q (A must contain only one such element):

```
I ← <FIRST A TO LAST A>;
X ← IF A ↓ 'I' = Q THEN 'I' ELSE NIL;
```

The special value UNDEFINED results when an operation which is syntactically correct has no meaning, such as

X ↓ 2 when X is a number. Any operation having one or more operands UNDEFINED yields an UNDEFINED result.

COMMENTS

A comment may occur at any place in a program. Any text beginning with the reserved word COMMENT and ending with a semicolon (;) is considered to be a comment, and is ignored.

OPERATOR PRECEDENCE

Operators in higher classes are executed first. Within classes, operators are executed from left to right as they occur in the program.

function calls OLD

↓ ↓...OF

FIRST LAST ROUND FLOOR CEIL ABS + * AND OR (unary)

* / MOD

+ - (binary)

- (unary)

= ¬= > >= < <=

NOT

AND

OR

WITH...←

IF...THEN...ELSE

←

LIST OF RESERVED WORDS

BEGIN	IF	AND	FOR
END	THEN	OR	STEP
READ	ELSE	NOT	UNTIL
WRITE	WITH	MOD	WHILE
NIL	OF	ROUND	DO
TUPLE	FIRST	FLOOR	INITIAL
DECLARE	LAST	CEIL	OLD
FUNCTION	TO	ABS	TRUE
DEFINE	BY	COMMENT	FALSE
	IS		

BNF DESCRIPTION OF SAMPLE SYNTAX

Note: The convention that all text enclosed between COMMENT and ; is ignored is not included in this description.

```
<program> ::= <block> .
<block> ::= BEGIN <block head> <statement list> END
<block head> ::= <declaration>
               | <block head> <function defn>
<declaration> ::= DECLARE <name list> ;
                | DECLARE <name list> , FUNCTION <name list> ;
<function defn> ::= DEFINE <name> ( <name list> ) ;
                          <function body> END
                  | DEFINE <name> ; <function body> END
<name list> ::= <name>
              | <name list> , <name>
<function body> ::= <declaration> <statement list> <expr>
                  | <statement list> <expr>
                  | <expr>
<statement list> ::= <statement>
                   | <statement list> <statement>
<statement> ::= <block>
              | <loop>
              | READ <read list> ;
              | WRITE <write list> ;
              | <left part> IS TUPLE (<num expr>,<num expr>) ;
              | <left part> ← <expr> ;
```

```
<read list> ::= <read atom>
              | <read list> , <read atom>
<read atom> ::= ( <left part> , <num expr> )
<write list> ::= <write atom>
               | <write list> , <write atom>
<write atom> ::= ( <expr> , <num expr> )
<left part> ::= <name>
              | <bounded left part>
              | <unbounded left part>
<bounded left part> ::= <name> <subscript> OF <num expr>
                      | <name> <subscript> OF ( <num expr>
                                , <num expr> )
                      | <bounded left part> <subscript>
                                OF <num expr>
                      | <bounded left part> <subscript> OF
                                ( <num expr> , <num expr> )
<unbounded left part> ::= <leader> <subscript>
<leader> ::= <name>
           | <leader> <subscript>
<loop> ::= FOR <name> ← <num expr> STEP <num expr> UNTIL
               <num expr> DO <init> ; <statement list> END
         | FOR <name> ← <num expr> UNTIL <num expr> DO
               <init> ; <statement list> END
         | WHILE <num expr> DO <init> ; <statement list> END
<init> ::= INITIAL <init atom>
         | <init> , <init atom>
<init atom> ::= <name> ← <expr>
```

```
<expr> ::= <num expr>
         | <expr> WITH <num expr> ← <num expr>
<num expr> ::= <logical expr>
             | IF <num expr> THEN <num expr> ELSE <num expr>
<logical expr> ::= <logical term>
                 | <logical expr> OR <logical term>
<logical term> ::= <logical factor>
                 | <logical term> AND <logical factor>
<logical factor> ::= <relation>
                   | NOT <relation>
<relation> ::= <arith expr>
             | <arith expr> = <arith expr>
             | <arith expr> ¬= <arith expr>
             | <arith expr> < <arith expr>
             | <arith expr> <= <arith expr>
             | <arith expr> > <arith expr>
             | <arith expr> >= <arith expr>
<arith expr> ::= <term>
               | - <term>
               | <arith expr> + <term>
               | <arith expr> - <term>
<term> ::= <factor>
         | <term> * <factor>
         | <term> / <factor>
         | <term> MOD <factor>
```

```
<factor> ::= <quantity>
           | FIRST <quantity>
           | LAST <quantity>
           | ROUND <quantity>
           | FLOOR <quantity>
           | CEIL <quantity>
           | ABS <quantity>
           | + <quantity>
           | * <quantity>
           | AND <quantity>
           | OR <quantity>
           | <number>
           | <unlabelled tuple>
<quantity> ::= <primary>
             | <tuple element>
<primary> ::= <name>
            | ' <name> '
            | OLD <name>
            | <function call>
            | ( <expr> )
            | NIL
            | TRUE
            | FALSE
<function call> ::= <name> ( <simple tuple> )
<tuple element> ::= <primary> <subscript>
                  | <tuple element> <subscript>
<subscript> ::= ↓ <primary>
              | ↓ <number>
```

```
<unlabelled tuple> ::= < <simple tuple> >
                     | < <tuple specifier> >
                     | < >
<simple tuple> ::= <expr>
                 | <simple tuple> , <expr>
<tuple specifier> ::= <num expr> TO <num expr>
                    | <num expr> TO <num expr> BY <num expr>
<name> ::= <letter>
         | <name> <letter>
         | <name> <digit>
<number> ::= <unsigned number>
           | <sign> <unsigned number>
<unsigned number> ::= <real>
                    | <real> <exponent>
                    | <exponent>
<real> ::= <integer>
         | <integer> .
         | . <integer>
         | <integer> . <integer>
<exponent> ::= E <integer>
             | E <sign> <integer>
<integer> ::= <digit>
            | <integer> <digit>
<letter> ::= A | B | C | . . . | X | Y | Z
<digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
```

PROGRAMMING EXAMPLES

1. Matrix Multiplication Program

This program reads matrices A and B and writes their product C. Input matrices A and B may be of any size and shape, provided that the number of columns of A is equal to the number of rows of B. Both input and output matrices are assumed to be represented as nested tuples, with one interior tuple representing each row. Rows and columns are assumed to begin with the subscript 1.

The program computes each product element C ↓ I ↓ J as the sum over K of the partial products A ↓ I ↓ K * B ↓ K ↓ J . The execution of the program might proceed as follows:

1. The input matrices A and B are read.
2. The following quantities are computed:

   L = the number of rows of A.

   M = the number of columns of A, which must equal the number of rows of B.

   N = the number of columns of B.
3. The tuples I, J, and K are assigned to serve as variables of iteration.
4. In a single statement, all the multiplication operations are performed and the results are stored in a triply-nested tuple T. If A and B are both square, N x N matrices, this single statement causes N cubed multiplications to occur

simultaneously (subject to the limitations of the hardware on which the problem is run).

5. Each product element C ↓ I ↓ J is computed as the sum of the elements of the tuple T ↓ I ↓ J, which represents summation over the variable K. A single statement causes this summation to occur for every element C ↓ I ↓ J simultaneously, completely defining the product matrix C.
6. The result C is written into the output medium.

```
BEGIN
    DECLARE A, B, C, I, J, K, L, M, N, T;
    READ (A,1), (B,2);
    L ← LAST A;
    M ← LAST (B ↓ 1);
    N ← IF LAST B = LAST (A ↓ 1) THEN LAST B ELSE NIL;
    I ← <1 TO L>;
    J ← <1 TO M>;
    K ← <1 TO N>;
    T ↓ 'I' OF (1,L) ↓ 'J' OF (1,M) ↓ 'K' OF (1,N) ←
       A ↓ 'I' ↓ 'K' * B ↓ 'K' ↓ 'J';
    C ↓ 'I' OF (1,L) ↓ 'J' OF (1,M) ← + T ↓ 'I' ↓ 'J';
    WRITE (C,3);
END.
```

2. Matrix Reduction by Gaussian Elimination

The matrix A is reduced to upper diagonal form by row operations. Rows and columns are assumed to begin with subscript 1. At every step, rows are shuffled so that the row with largest coefficient is used as the pivotal row. A may be a matrix of any size and shape.

```
BEGIN
  DECLARE A, B, C, D, I, J, K, L, M, COLUMN, PIVNO, PIVOT,
        FACTOR, FUNCTION MAXNO;
  DEFINE MAXNO(U);
     COMMENT: RETURNS THE SUBSCRIPT NUMBER OF THE ELEMENT OF
        U HAVING GREATEST ABSOLUTE VALUE;
     DECLARE IU, V;
     FOR IU ← FIRST U UNTIL LAST U DO
        INITIAL V ← FIRST U;
        V ← IF ABS(U ↓ (OLD V)) > ABS(U ↓ IU)
           THEN OLD V ELSE IU;
     END
     V
  END
```

```
  READ (A,1);
  L ← LAST A;
  FOR I ← 1 UNTIL L - 1 DO
     COMMENT: ONCE FOR EACH ROW EXCEPT THE LAST;
     INITIAL D ← A;
     J ← <1 TO I>;
     K ← <I+1 TO L>;
     M ← <I TO L>;
     COMMENT: ISOLATE THE ITH COLUMN;
     COLUMN ↓ 'M' OF (I,L) ← OLD D ↓ 'M' ↓ I;
     COMMENT: FIND THE PIVOTAL ROW;
     PIVNO ← MAXNO(COLUMN);
     PIVOT ← COLUMN ↓ PIVNO;
     COMMENT: INTERCHANGE PIVOTAL ROW WITH ITH ROW;
     B ← OLD D WITH I ← OLD D ↓ PIVNO;
     C ← B WITH PIVNO ← OLD D ↓ I;
     COMMENT: DIVIDE EACH ROW BY THE APPROPRIATE FACTOR;
     FACTOR ↓ 'K' OF (I+1,L) ← C ↓ 'K' ↓ I / PIVOT;
     D IS TUPLE(1,L);
     D ↓ 'J'  ← C ↓ 'J';
     D ↓ 'K'  ← C ↓ 'K' - C ↓ I * FACTOR ↓ 'K';
  END
  WRITE (D,2);
END.
```

3. Universal Turing Machine Simulator

This program reads a tuple of 5-tuples which define the Turing Machine to be simulated. Each 5-tuple is a rule representing a combination of current state, current symbol, next state, next symbol, and movement along the tape. The program also reads the initial tape, state, and position of the machine. It then simulates the machine, stopping when the current state and letter do not match any 5-tuple. The tape is allowed to grow without bound; squares to the right and left of the initially defined tape are assumed to contain all zeros. After each Turing Machine cycle, the cycle number, state, position, and complete tape are sent to the output medium. This is not a good parallel program, but it shows the universality of the language.

```
BEGIN
  DECLARE RULES, INITTAPE, INITSTAT, INITJ, TAPE, STATE,
     COMMAND, LETTER, I, J, K, FUNCTION CATLEFT, CATRIGHT;
  DEFINE CATLEFT(TL, CL);
     COMMENT: RETURNS THE TUPLE TL, WITH THE ADDITIONAL
         ELEMENT CL CONCATENATED AT THE BEGINNING OF THE
         TUPLE (USED WHEN THE TAPE GROWS TO THE LEFT);
     DECLARE IL, QL;
     IL ← <FIRST TL TO LAST TL>;
     QL IS TUPLE(FIRST TL - 1, LAST TL);
     QL ↓ 'IL' ← TL ↓ 'IL';
     QL ↓ (FIRST TL - 1) ← CL;
     QL
  END
```

```
DEFINE CATRIGHT(TR, CR);
   COMMENT: RETURNS THE TUPLE TR, WITH THE ADDITIONAL
        ELEMENT CR CONCATENATED AT THE END OF THE
        TUPLE (USED WHEN THE TAPE GROWS TO THE RIGHT);
   DECLARE IR, QR;
   IR ← <FIRST TR TO LAST TR>;
   QR IS TUPLE(FIRST TR, LAST TR + 1);
   QR ↓ 'IR' ← TR ↓ 'IR';
   QR ↓ (LAST TR + 1) ← CR;
   QR
END
READ (RULES, 1), (INITTAPE,2), (INITSTAT,3), (INITJ,4);
COMMENT:
      RULES IS A TUPLE OF 5-TUPLES WHICH GOVERN THE
          BEHAVIOR OF THE MACHINE.
      INITTAPE IS A TUPLE REPRESENTING THE INITIAL TAPE.
      INITSTATE IS A NUMBER REPRESENTING THE INITIAL
          STATE OF THE MACHINE.
      INITJ IS A NUMBER REPRESENTING THE INITIAL LOCATION
          OF THE MACHINE (AS A SUBSCRIPT INTO INITTAPE);
K ← <0 TO LAST TUPLES>;
WHILE TRUE DO
   INITIAL TAPE ← INITTAPE,
           STATE ← INITSTAT,
           J ← INITJ,
           LETTER ← INITTAPE ↓ INITJ,
           I ← 0;
```

```
    I ← OLD I + 1;
    COMMENT: FIND THE 5-TUPLE WHICH MATCHES THE CURRENT
       SITUATION;
    COMMAND ← IF RULES ↓ 'K' ↓ 0 = OLD STATE
       AND RULES ↓ 'K' ↓ 1 = OLD LETTER
       THEN RULES ↓ 'K' ELSE NIL;
    COMMENT: CHANGE STATE AND POSITION, AND WRITE ON TAPE;
    STATE ← COMMAND ↓ 2;
    J ← OLD J + COMMAND ↓ 4;
    TAPE ← IF J > LAST OLD TAPE THEN
       CATRIGHT(OLD TAPE WITH OLD J ← COMMAND ↓ 3, 0)
       ELSE IF J < FIRST OLD TAPE THEN
       CATLEFT(OLD TAPE WITH OLD J ← COMMAND ↓ 3, 0)
       ELSE OLD TAPE WITH OLD J ← COMMAND ↓ 3;
    LETTER ← TAPE ↓ J;
    COMMENT: PRINT THE RESULTS OF THIS CYCLE;
    WRITE (I,4*I-3), (STATE,4*I-2), (J,4*I-1), (TAPE,4*I);
  END
END.
```

4. Sort Program Using "Divide and Conquer"

This program sorts a tuple recursively by splitting it into two parts, sorting each part, and merging the results. It was written to illustrate the recursive properties of the language.

```
BEGIN
    DECLARE U, V, FUNCTION SORT, MERGE;
    DEFINE SORT(X);
       COMMENT:  IF X HAS 1 ELEMENT, RETURN IT.
            IF X HAS 2 ELEMENTS, SORT THEM AND RETURN THEM.
            IF X HAS 3 OR MORE ELEMENTS, BREAK IT IN HALF,
                CALL SORT FOR EACH HALF, AND CALL MERGE
                FOR THE RESULTS;
       DECLARE H, M, N, W, Y, Z;
       W ↓ 0 OF 1 ← IF X ↓ 0 < X ↓ 1 THEN X ↓ 0 ELSE X ↓ 1;
       W ↓ 1 OF 1 ← IF X ↓ 0 < X ↓ 1 THEN X ↓ 1 ELSE X ↓ 0;
       H ← CEIL(LAST X/2);
       M ← <0 TO H-1>;
       N ← <H TO LAST X>;
       Y ↓ 'M' OF (H-1) ← X ↓ 'M';
       Z ↓ ('N'-H) OF (LAST X-H) ← X ↓ 'N';
       COMMENT: THE FOLLOWING EXPRESSION IS THE VALUE TO BE
            RETURNED;
       IF LAST X = 0 THEN X
          ELSE IF LAST X = 1 THEN W
          ELSE MERGE(SORT(Y), SORT(Z))
    END
```

```
DEFINE MERGE(A,B);
   COMMENT: TUPLES A AND B ARE EACH IN NUMERIC ORDER.
        MERGE THEM INTO A SINGLE TUPLE, ALSO IN
        NUMERIC ORDER;
   DECLARE I, J, K, C;
   FOR K ← 0 UNTIL (LAST A + LAST B + 1) DO
      INITIAL I ← 0;
      INITIAL J ← 0;
      I ← IF A ↓ (OLD I) < B ↓ (OLD J)
           THEN OLD I + 1 ELSE OLD I;
      J ← IF A ↓ (OLD I) >= B ↓ (OLD J)
           THEN OLD J + 1 ELSE OLD J;
      C ↓ K OF LAST A + LAST B + 1 ←
         IF A ↓ (OLD I) < B ↓ (OLD J) THEN A ↓ (OLD I)
         ELSE B ↓ (OLD J);
   END
   C
END
READ (U,1);
V ← SORT(U);
WRITE (V,2);
END.
```

CHAPTER THREE

THE SYSTEM

ORGANIZATION

Processing of a SAMPLE program consists of two phases: compilation and execution. Compilation, in turn, consists of two steps: scanning and parsing. The processes of scanning, parsing, and execution operate on the following four passive storage devices:

1. The Text Storage Unit (TSU)
2. The Memory
3. The Instruction Store (IS)
4. The Ready List (RL)

The scanner accepts the source program as a string of characters. Scanning through the program from left to right, the scanner recognizes reserved words, names, and other syntactic entities, converts them to symbols, and places them in the Text Storage Unit.

The input to the parser is the output of the scanner: a string of symbols in the TSU which represent the source program. The parser performs "reductions" on these symbols according to the rules of the SAMPLE grammar, meanwhile emitting the object program into the Memory, Instruction

Store, and Ready List. At the conclusion of parsing, the TSU will contain the single goal symbol <program>, and the other storage entities will contain the object program, ready for execution.

When the program is ready for execution, the Instruction Store (IS) contains a number of instructions, each of which has an opcode and up to four operands. Most of the operands are addresses of data in the Memory. The instructions in the IS do not necessarily appear in the order in which they will execute. The Ready List (RL) contains copies of those instructions which are currently ready to be executed. Each cell in the Memory contains, in addition to a data value, a pointer to a list of instructions in the IS which depend on that particular data value as an input operand. The Memory cell contains the address of one such instruction, and the instructions form a linked list, each one pointing to the next.

The actual execution is performed by many processors, all active simultaneously. All processors continuously execute the same sequence of actions, called the Basic Instruction Routine, which has the following steps:

1. The processor obtains from the Ready List an instruction which is ready to be executed.
2. The processor fetches the input operands from the memory, performs the indicated operation, and writes the resulting output operand in memory.

3. The processor now follows the linked list of instructions which are waiting for the memory cell which has just been defined. In each such instruction, the ready bit is turned on for the operand in question. If any such instruction becomes ready (all its ready bits are on), it is added to the Ready List. When the processor reaches the end of the list of waiting instructions, it returns to step (1) above.

Execution is complete when all processors are idle and the Ready List is empty.

During execution, many processors are making simultaneous access requests to the Memory, IS, and RL. In order to facilitate parallelism, each of these storage units is organized into several banks, and the addresses of cells within the unit are interleaved among the banks. Each storage unit can simultaneously service multiple access requests if the cells to be accessed lie in different banks. Thus many processors can be active with a minimum of storage conflicts.

The four storage units are described in detail below. The processes of compilation and execution are described in detail, with examples, in the following chapters.

THE TEXT STORAGE UNIT (TSU)

The Text Storage Unit accepts the source program from the scanner, and contains the partially-parsed program at all times during parsing. The program is represented in the TSU as an ordered string of symbols from the SAMPLE grammar. This thesis does not specify the means by which the scanning and parsing algorithms are to be implemented; therefore we will not specify the detailed structure of the TSU. One possibility is that compilation of SAMPLE programs might be done on a conventional machine, and the resulting object program executed on the SAMPLE system. In this case, the TSU is simply the memory of the conventional machine.

However it is implemented, the TSU must contain an ordered sequence of symbols, and the sequence must be able to be modified during the parsing process. Each symbol must contain the following information:

1. A "type" field, which denotes one of the symbol types of the SAMPLE grammar.
2. A "value" field, which contains a reference to the value of the symbol. This might be:
   a. A memory address (M-reference).
   b. A tag for which no memory cell has yet been allocated (T-reference).
   c. The address of an instruction in the IS (I-reference).

3. A "code begin" field which points to the first of a linked list of instructions in the IS which are associated with the symbol.
4. A "code end" field which points to the last of the linked list of instructions in the IS which are associated with the symbol.

In the following chapters, it will occasionally be necessary to represent a TSU symbol. Terminal symbols will be represented by themselves, such as + . Nonterminal symbols will be represented by their "type", "value", and "code begin" fields enclosed in brackets, such as <term.T1.I2> . If the content of one or more of these fields is immaterial, it may be omitted; for example, the above symbol might have been written as <term.T1> or even as <term> .

THE MEMORY

The Memory is the storage place for data during the execution of a program. Its entries, called "cells", are addressable by ordinal numbers. Memory addresses are referred to as "M-references". Each memory cell contains:

1. A "type" bit, which denotes whether the cell contains a number or a tuple.
2. A "ready" field, which can denote one of eight levels of readiness. The levels of readiness correspond to levels of nesting of loops. Level

seven denotes the main program, level six a simple loop, level five a nested loop, etc. The zeroth level of readiness denotes "not ready on any level". For a tuple to be marked "ready", only its "first", "last", and "start" fields must be defined; none of its elements need be defined.

3. A "content" field.
   a. If the cell is a number, the "content" field contains a real number or the special value UNDEFINED.
   b. If the cell is a tuple, the "content" field contains:
      1. A "first" field equal to the first subscript number of the tuple.
      2. A "last" field equal to the last subscript number of the tuple. (If the content of the cell is the null tuple, the "first" and "last" fields contain a special code.)
      3. A "start" field which points to the memory address containing the first tuple element. All tuple elements must occupy contiguous cells in memory, beginning with the cell pointed to by the "start" field. Each tuple element may be either a number or a tuple. If the tuple has no elements, the "start" field contains a special "null" code.

4. An "old" field, which points to another memory cell. In loop processing, this field points to the value of this cell the last time around the loop.
5. A "link" field, which points to an instruction in the IS which depends on the memory cell as an input operand. This instruction is the first of a linked list of dependent instructions which must be updated when the memory cell becomes ready. The "link" field may contain the null pointer.

The memory is organized into banks, each of which can independently service an access request in one cycle. The memory addresses are interleaved among the banks in such a way that successive addresses are in different banks. Requests for a particular memory address are automatically routed to the correct bank. The memory can service the following types of requests (request parameters shown in parentheses):

1. Memory Read (MR (address))
   The contents of cell (address) are sent to the requesting processor. The cell is unchanged.

2. Memory Read-Write (MRW (address) (data))
   The contents of cell (address) are sent to the requesting processor, and (data) replaces all the fields of the memory cell except the "link" field, which remains unchanged.

3. Memory Read-Write Link Conditional

(MRWLC (address) (data) (level))

The contents of cell (address) are sent to the requesting processor. The "ready" field of the memory cell at (address) is compared to (level). If the memory cell is ready on the same or a lesser level than (level), the cell is unchanged; otherwise (data) replaces the "link" field of the cell, and its other fields are unchanged.

4. Memory Allocate (MA (number))

(Number) new, unused memory cells are allocated to the requesting processor, in a connected block. The processor is given the address of the first cell in the block. During the run of a program, memory cells are always allocated in order of ascending addresses. A central allocation register contains the address of the first non-allocated memory cell. Servicing a Memory Allocate request simply consists of returning the content of the allocation register to the requesting processor, and incrementing the allocation register by (number). Since the allocation register can be a fast electronic register, we assume that many Memory Allocate requests can be serviced in one cycle. Memory Allocate requests do not make any memory bank busy. It is assumed that all the newly-allocated memory cells are marked "not ready on any level".

THE INSTRUCTION STORE (IS)

Instructions are placed in the Instruction Store by the compiler, and are modified and executed by the processors. The IS entries are addressable by ordinal numbers. Instruction addresses are referred to as "I-references". Each instruction contains:

1. A "level" field, which can denote one of eight levels, corresponding to loop nesting levels.
2. An "opcode", which denotes a particular operation, or may be one of the following special codes:
   a. Continuation of previous opcode
   b. ITERANT
   c. PARAMETER
   d. RESULT
   e. INTERNAL
   f. DOUBLE
   g. LOCAL
   h. STARTUP
   i. NESTED
3. Four operands, each of which contains:
   a. A reference to the actual operand. This may be:
      1. A memory address (M-reference).
      2. A tag for which no memory cell has yet been allocated (T-reference).
      3. The address of another instruction (I-reference).

4. A "literal", representing an integer value (L-reference).
5. Unused (instruction has fewer than four operands).

b. A "ready" bit, denoting that the operand cell is ready on at least the same level as the level of the instruction. If the operand is an I- or L-reference or unused, it is always considered to be ready and its ready bit is automatically set.

c. A "link" field which points to another instruction. By means of these link fields, linked lists are formed of instructions which are waiting for a particular memory cell as an input operand. The link field of the memory cell points to the first such instruction, and each instruction points to the next instruction by means of the link field associated with the particular operand in question. Because each instruction has four operands, it can participate in up to four linked lists. A link field may contain the null pointer.

4. A "statement link" field which points to another instruction. By means of this field, instructions resulting from the same source language statement form a linked list for processing by the EXPAND instruction. This field may contain the null pointer.

5. A "presence" bit. When the cell is allocated but does not yet contain an instruction, this bit is set to 0 ("not present"). After an instruction has been placed in the cell, the bit is set to 1 ("present").

The IS is organized into banks, each of which can independently service an access request in one cycle. The instruction addresses are interleaved among the banks in such a way that successive addresses are in different banks. Requests for a particular address are automatically routed to the correct bank. The IS can service the following types of requests (request parameters shown in parentheses):

1. Instruction Read (IR (address))
   The instruction at (address) is sent to the requesting processor.

2. Instruction Write (IW (address) (data))
   (Data) becomes the new IS entry at (address). The "presence" bit is part of the data which is written; this request usually sets this bit to "present".

3. Instruction Set Ready Bit
   (ISRB (address) (data) (level))
   The instruction at (address) is sent to the requesting processor. In addition, the operands of

the instruction are all compared with (data); all operand references which match (data) have their ready bits turned on if (level) is greater than or equal to the "level" field of the instruction.

4. Instruction Set Ready Bit, Low Priority
(ISRBL (address) (data) (level))
This request behaves exactly the same as ISRB except that it has lower priority than any other request. It is used by a processor which is trying to do an ISRB on an instruction which is not yet present. The low priority of the ISRBL request prevents it from "locking out" any processor which may be trying to write an instruction into (address).

5. Instruction Allocate (IA (number))
(Number) new, unused instruction cells are allocated to the requesting processor, in a connected block. The processor is given the address of the first cell in the block. During the run of a program, instruction cells are always allocated in order of ascending addresses. A central allocation register contains the address of the first non-allocated instruction cell. Servicing an Instruction Allocate request simply consists of returning the content of the allocation register to the requesting processor, and

incrementing the allocation register by (number). Since the allocation register can be a fast electronic register, we assume that many Instruction Allocate requests can be serviced in one cycle. Instruction Allocate requests do not make any instruction bank busy. It is assumed that the presence bits of all newly-allocated instructions are set to "not present".

THE READY LIST (RL)

The Ready List contains those instructions which are ready to be executed because all their operands have been found to be ready. The entries are not addressable. Each entry consists of an opcode and eight operands, each of which may be an M-, I-, or L-reference, or unused.

The RL is organized into banks. Because Ready List requests do not refer to specific addresses, any bank may service any request, subject to certain constraints mentioned below. A unique but random ordering ensures that each incoming request in a given cycle is serviced by a different bank until all banks are busy. The RL can service the following types of requests:

1. Readylist Add (RA (data))

   This request can be serviced only by a bank which is not already full. The (data), consisting of an

instruction ready to be executed, is added to the Ready List. However, in the case when all banks of the RL are empty and another processor is requesting a simultaneous Readylist Fetch (RF), the (data) is sent directly to the other processor rather than to the Ready List.

2. Readylist Fetch (RF)

This request can be serviced only by a bank which is not completely empty. The bank sends one of its ready instructions to the requesting processor, and deletes this entry from the bank. Each Ready List bank serves as a first-in, first-out queue of ready instructions, responding to each RF request by returning the instruction which has been present in the bank for the longest time. If all banks of the RL are completely empty and another processor is requesting a simultaneous Readylist Add (RA), the other processor sends its instruction address directly to the requesting processor.

CHAPTER FOUR

COMPILATION

THE SCANNER

The purpose of the scanner is to convert the SAMPLE source program from a string of alphabetic and numeric characters to a string of symbols which are meaningful to the recognizer. In so doing, the scanner also removes comments and block structure from the program, allocates memory cells to names, and performs certain other operations.

Input to the scanner is the SAMPLE program as written by the programmer. The scanner makes a single sequential pass over the program. During this pass, "bindings" are made according to the following rules:

1. When the scanner reads a declaration which is not inside a function definition, it binds a unique memory address (M-reference) to each declared variable name, and a unique instruction address (I-reference) to each declared function name. The scanner must request allocation of empty cell addresses from the memory and the instruction store for this purpose.

2. When the scanner reads a formal parameter list in a function definition, it binds a unique tag (T-reference) to each parameter. In addition, it creates in the IS a linked list of instruction cells, each containing one of the newly-bound T-references, with the label "parameter" in the opcode field. The T-references must appear on the linked list in the same order as they appeared in the parameter list. The symbol <name list.Ip> is emitted, where Ip is the starting address of the linked list.
3. When the scanner reads a declaration in a function body (or in any block nested inside a function body), it binds a unique tag (T-reference) to each declared variable name, and a unique, newly-allocated instruction address (I-reference) to each declared function name. In addition, the scanner makes additions to a linked list of instruction cells which begins at the I-reference bound to the name of the function inside whose body the declaration is found. To this list is added all the T-references bound to the newly-declared names, in cells labelled "internal". In the case of a declaration occurring inside a function which is nested inside another function, the T-references are added to the "internal" list of the innermost function only. Thus, each function name is bound to a linked list of instruction cells labelled

"internal", containing T-references for all variables declared inside the function.

Whenever the scanner encounters a declared name or function parameter which duplicates a name which was declared or appeared as a function parameter in an outer block, the old reference bound to this name is "pushed down" and a new reference binding is made. The new binding is considered active until the scanner comes to the end of the block in which the binding was made; at this time, the new binding is deactivated and the old reference is once again bound to the name. Since a name can be declared repeatedly in nested blocks, the scanner must, in general, remember the references bound to a given name at each lexic level from the outermost block to the block which is currently being scanned. This forces the scanner to be sequential in nature and to contain a "pushdown stack" structure to record the history of reference bindings for each name. The scanner is the only necessarily sequential process in the SAMPLE system.

As the scanner makes its single pass through the program, it emits a sequence of symbols into the Text Storage Unit. Each symbol emitted consists of a symbol type code, and may or may not also contain an M-, T-, or I-reference. The symbols are emitted and placed into the TSU in order as they are recognized by the scanner. The following rules for emitting symbols are observed by the scanner:

1. Blanks are ignored, except insofar as they delimit other symbols. For example, a blank is required between X and IS and between IS and TUPLE in the bounding statement X IS TUPLE(0,5);
2. Comments are ignored; whenever the scanner recognizes the reserved word COMMENT, it skips over all characters until the next semicolon without emitting any symbols.
3. Declarations cause no symbols to be emitted, but update the internal binding tables of the scanner. If a declaration is inside a function definition, it may cause additional entries to be made to the "internal" list of the function.
4. When a parameter list is encountered in a function definition, a linked "parameter list" is created in the IS, and the symbol <name list.Ip> is emitted, where Ip is the starting address of the linked list.
5. When an operator, punctuation symbol, or reserved word is encountered, a symbol is emitted whose symbol type denotes the particular symbol encountered.
6. When a name (any sequence of characters beginning with a letter, which is not a reserved word) is encountered, a symbol of type <name> is emitted, containing the M-, T-, or I-reference bound to the name in the block which is currently being scanned.
7. When a constant number (such as -5 or 2.6E8) is encountered, the scanner requests allocation of an empty memory cell, and places in the cell the system's

internal representation for the given constant. The memory cell is marked ready on level 0. A symbol of type <number> is then emitted, containing an M-reference equal to the address of the memory cell containing the constant.

THE PARSER

After the scanning process is complete, the remainder of the compilation consists of recognizing reducible strings of symbols in the TSU and performing the appropriate reductions according to the SAMPLE grammar, while emitting appropriate instructions into the Instruction Store.

There exist standard algorithms, such as McKeeman's Extended Precedence Method (29) (30), for determining whether a given string of symbols should be reduced, by examining nearby context symbols. These algorithms demand that the program be parsed by a sequential left-to-right scan. In conventional languages, a sequential left-to-right parse is also necessitated by two other considerations:

1. The code generated during the parse must be emitted in left-to-right order so that it can be executed properly.
2. The parser must make use of context-sensitive information, pertaining to declarations and block structure, which can only be obtained by a left-to-right scan of the entire program.

However, in the SAMPLE system, both of these considerations have been eliminated. The machine instructions are sequenced by their data dependencies rather than by their physical ordering, and all context-sensitive information has been filtered out of the SAMPLE program by the scanner before the parsing process begins. Therefore, if an algorithm could be found for deciding whether to reduce a given string of symbols, without requiring that all symbols to the left of the given string already be parsed, the parsing of SAMPLE programs could be treated as a parallel process. Several parsing processors could simultaneously be active at different places in the TSU, independently performing reductions and emitting instructions into the Instruction Store.

The reductions of the SAMPLE grammar are listed in Appendix A, together with the actions to be taken by the compiler when each reduction is performed. The decision to make the reduction may be made either by a conventional, left-to-right parsing algorithm, or by an unconventional, parallel parsing algorithm. In every reduction, it is implicit that the symbols on the right side will be replaced in the TSU by the symbol on the left. When compilation is complete, the Text Storage Unit will contain the single symbol <program>, and the Memory, Instruction Store, and READY List will contain the compiled code for the program.

EXAMPLE

We will now consider how the compiler might treat the simple program we used as an exalmple in Chapter Two. The program is repeated below:

```
BEGIN
    W ← A-B;
    X ← (A+B)/W;
    Y ← (A*B)/W;
    Z ← X>Y;
END
```

Before we can understand the action of the compiler, we must consider how the compiler treats iterated statements. Iteration denoted by the quoted-tuple-name convention described in Chapter 2 calls for all copies of the iterated statement to be executed in parallel, if possible. In order to accomplish this purpose, the SAMPLE system physically produces multiple copies of the machine instructions which result from the iterated statement. However, since the number of iterations may be data-dependent, it is not known at compile-time how many copies of the machine instructions must be made. Therefore, the compiler produces, for each machine instruction, a template from which multiple copies may be produced at run-time. The template contains actual memory addresses

(M-references) for those operands identified by name in the original source program, because each name in the source program is uniquely mapped into a single memory cell, regardless of its appearance in iterated statements. However, for operands which represent compiler-generated intermediate results (such as the quantity (A+B) in our example), the template contains only a tag (T-reference). Only at run time will memory cells be allocated for the T-references, and at this time, depending on the nature of the iteration, possibly many memory cells may be allocated for a single T-reference.

The machine instruction which expands the templates into multiple new machine instructions, allocates memory cells for the T-references, and replaces the T-references by real memory addresses in the newly-generated instructions, is the EXPAND instruction. The output of the compiler for a statement in the original source program is the following:

1. A linked list of instruction templates, linked together by their "statement link" fields.
2. An EXPAND instruction which has the above list as its operand, and which will generate real instructions from it at run-time.

The example program above contains four statements. Each statement is compiled into an EXPAND instruction and a linked list of templates, as explained above. In our simple case, none of the statements involves iteration, and

so no additional information is needed before we can expand the templates; therefore, all the EXPAND instructions are judged to be ready, and are placed in the Ready List by the compiler.

The results of compilation of our example are shown in Figure 4.1. The memory addresses allocated for the names A, B, W, X, Y, and Z have been denoted Ma, Mb, Mw, Mx, My, and Mz, respectively. Only memory cells Ma and Mb are assumed to be defined (ready) at the beginning of processing. The two T-references generated by the compiler to denote intermediate results in the templates are T1 and T2. The four statements in the source program resulted in linked lists of instruction templates of length one, two, two, and one, respectively. When execution begins, the four EXPAND instructions will all begin generating real instructions from the templates.

MEMORY

	Value	Link
(Ma)	A	
(Mb)	B	
(Mw)		
(Mx)		
(My)		
(Mz)		
(M1)		
(M2)		

INSTRUCTION STORE

	Opcode	Operands (R)=ready bit on	Stat. Link
(I1)	-	Mw( ),Ma( ),Mb( )	
(I2)	+	T1( ),Ma( ),Mb( )	
(I3)	/	Mx( ),T1( ),Mw( )	
(I4)	*	T2( ),Ma( ),Mb( )	
(I5)	/	My( ),T2( ),Mw( )	
(I6)	>	Mz( ),Mx( ),My( )	

READY LIST

EXPAND I1
EXPAND I2
EXPAND I4
EXPAND I6

FIGURE 4.1

EXAMPLE PROGRAM AFTER COMPILATION

CHAPTER FIVE

EXECUTION

PROCESSORS

The execution process is carried out in parallel by many processors. Each processor may have many ports through which it may make simultaneous requests of the Memory, Instruction Store, and Ready List (the types of requests which may be made have been described in detail in the SYSTEM chapter.) The Memory, IS, and RL operate on a synchronous basis; in each cycle, up to one service request per bank may be satisfied by each storage unit. A priority ordering is defined among the processors, and among the ports of each processor. When two ports make a request of the same storage bank, the lower-priority port is not satisfied until a later cycle. A processor may not proceed to make further requests until all its previous requests have been satisfied.

All the processors repeatedly execute a sequence of operations called the Basic Instruction Routine (BIR). The BIR is a several-step process, and at any given point in time the various processors may be at various points in executing the BIR. During execution of the BIR, a processor

may find it necessary to execute one or more of two auxiliary routines called the Readiness Routine and the Instruction Generate Routine.

Processors may be added to or deleted from the system at will without altering the basic structure of the system. Similarly, the number of request ports per processor may be altered at will.

The 48 machine instructions are listed and described in detail in Appendix B. The Basic Instruction Routine, Readiness Routine, and Instruction Generate Routine are described below.

THE BASIC INSTRUCTION ROUTINE

1. By means of a Readylist Fetch (RF) request, the processor obtains an instruction which is ready to be executed.
2. By appropriate Memory Read (MR) and Instruction Read (IR) requests, the processor reads all operands necessary to execute the instruction. This may require many sequential requests in the case of some instructions which operate on linked lists in the IS. It is assumed that each processor has sufficient internal storage to store operands and intermediate results during processing of an instruction, and that the access time of this internal storage is small compared to the access

time of the main storage devices which are external to the processor.

3. The processor executes the instruction. This may involve any of the following processes:

   a. One or more memory cells may be assigned values, or the readiness level of a memory cell may be increased. When this occurs, the processor executes the Readiness Routine (described below).

   b. One or more new instructions may be created and released for execution. When this occurs, the processor executes the Instruction Generate Routine (described below).

   c. New areas of memory may be allocated by means of Memory Allocate (MA) requests.

   d. New instructions may be placed in the IS but not released for execution. This is done by means of Instruction Allocate (IA) and Instruction Write (IW) requests.

   e. Existing instructions in the IS, which have not yet been released for execution, may be altered. This is done by means of Instruction Read (IR) and Instruction Write (IW) requests.

   f. Memory cells which were previously marked ready on a given level may be marked not ready on that level. This is done by means of Instruction

Read (IR) and Instruction Write (IW) requests. This action is taken only by certain instructions for the implementation of loops. No memory cell is marked not ready on a given level until all instructions which use that memory cell on that level have been executed; hence, no updating of dependent instructions is necessary when a memory cell is marked not ready.

THE READINESS ROUTINE

1. This routine is executed when one or more memory cells have their values newly defined by a processor, or when the readiness level of a memory cell is increased. The processor writes the new values into the memory cells by means of Memory Read-Write (MRW) requests. The "ready" fields of the memory cells are set to the level of readiness of the instruction which defined them. The "link" fields of the newly-defined cells are unchanged. The MRW requests read from memory the contents of the "link" fields of the newly-defined cells. For each newly-defined memory cell, the following is done:
2. The instruction pointed to by the "link" field of the newly-defined cell is updated by means of an Instruction Set Ready Bit (ISRB) request. This reads the instruction into the processor and, in

addition, sets the appropriate "operand ready" bit or bits in the copy of the instruction which remains in the IS, if the newly-defined operand is ready on at least the level of the instruction. If the "presence" bit of the instruction just read is set to "not present", the processor issues repeated Instruction Set Ready Bit, Low Priority (ISRBL) requests for the instruction until the "presence" bit is found to be set to "present". The processor then examines the instruction which it has read; if all its "operand ready" bits are on except the one associated with the newly-defined operand, it enters the instruction into the Ready List by means of a Readylist Add (RA) request. (Another Instruction Read (IR) request may be necessary to read the second half of a two-cell instruction.) (If all "operand ready" bits, including the one associated with the newly-defined operand, were already on, the instruction has already been added to the Ready List at an earlier time; hence, it is ignored.) In addition, the processor finds the "link" field of the instruction which is associated with the newly-defined operand (or the first such operand, it there are more than one). The processor then issues an ISRB request for the instruction pointed to by this "link" field, if any.

3. The processor continues in the above manner down the linked list of dependent instructions, issuing ISRB, ISRBL, and RA requests as needed, until it comes to a link field equal to "none". (This may occur before any instructions have been fetched.) This terminates the Readiness Routine. If the Readiness Routine is the last step in the Basic Instruction Routine, and if the last instruction on the dependency list is found to be ready, this instruction is not added to the Ready List but is retained, and a new Basic Instruction Routine is begun to execute it.

THE INSTRUCTION GENERATE ROUTINE

1. This routine is executed when one or more new instructions are to be released for execution. For each such instruction, the processor requests a new IS cell or pair of cells to be allocated by means of an IA request. These requests automatically set the "presence" bits of the allocated instruction cells to "not present". The addresses of the allocated cells are returned to the processor. Then, for each newly-released instruction, the following is done:
2. The instruction may have one or more input operands (M-reference operands whose "operand ready" bit is

not yet on). The processor issues a Memory Read-Write Link Conditional (MRWLC) request for each such operand. These requests read the operand cells, and, for each cell not yet ready on the level of the new instruction, its "link" field is made to point to the new instruction.

3. The processor then examines the memory cells which were read by the MRWLC requests. For each cell having its "ready" field equal to or greater than the level of the newly-released instruction, the corresponding "operand ready" bit of the instruction is turned on. All other input operands have had their "link" fields altered to point to the new instruction; hence, the old "link" fields of these cells, which were read by the MRWLC requests, are inserted into the "link" fields of the corresponding operands of the new instruction. Thus the continuity of the list of dependent instructions is preserved, and the new instruction is added at the head of the list.

4. The processor has now filled in all "operand ready" and "link" fields of its internal copy of the new instruction. If the instruction has any non-ready operands, the processor places the instruction into the IS cell provided for it by means of an Instruction Write (IW) request, which sets the "presence" bit of the instruction to "present". If

the instruction has all its operands ready, the processor sends it directly to the Ready List by means of a Readylist Add (RA) request. If the Instruction Generate Routine is the last step in the Basic Instruction Routine, and if a newly-released instruction is found to be ready, this instruction is not added to the Ready List but is retained, and a new Basic Instruction Routine is begun to execute it.

EXAMPLE

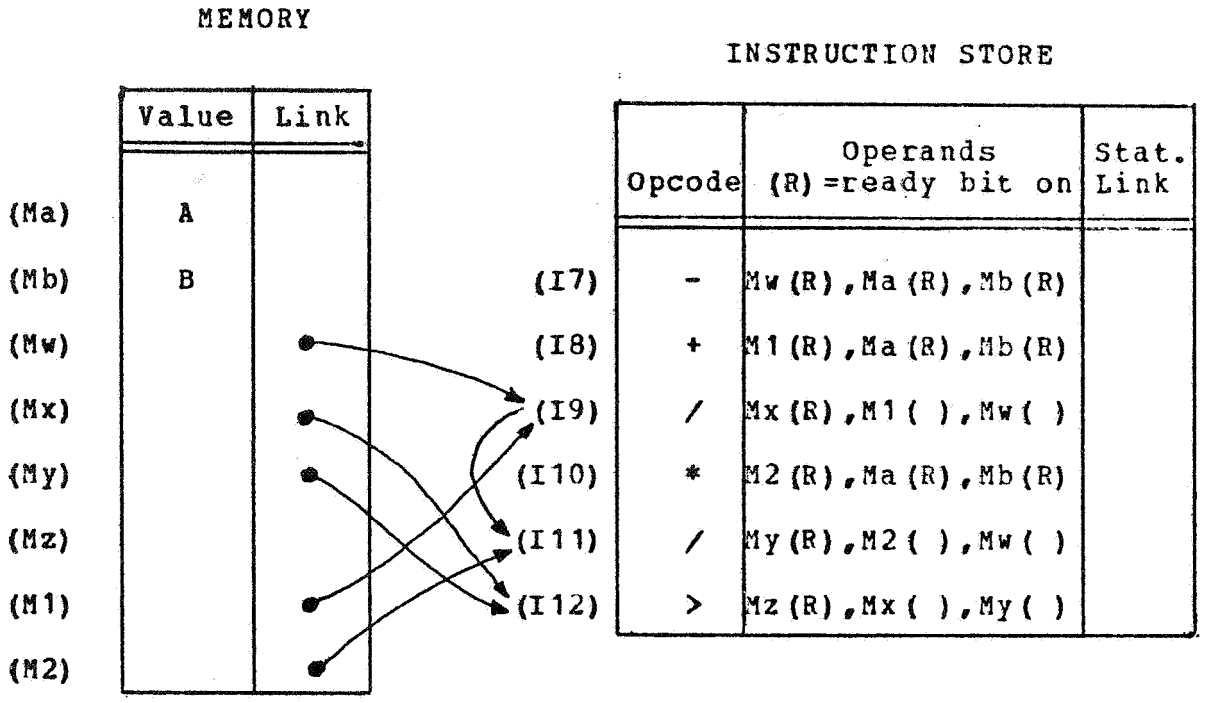
We will now study the execution of the program we used as an example in Chapters 2 and 4. (The reader should see Figure 4.1 for the state of the system before execution begins). We will consider the execution process to occur in two phases: expansion and processing. (In reality, these phases are overlapped.) In the expansion phase, the four EXPAND instructions, which are already known to be ready, are executed. The results of this phase are shown in Figure 5.1 and are described below:

1. The instruction templates I1 - I6 (not shown) are used to generate the real instructions I7 - I12. In our case, each template gives rise to exactly one instruction. (If any iterated statements had occurred in the source program, some templates would have yielded multiple instructions.)

2. Memory cells are allocated for storage of intermediate results (T-references in the compiled code). In our case, the T-references T1 and T2 are changed to memory addresses M1 and M2.
3. As the newly-generated instructions are released for execution, memory cell pointer fields are filled in, so that each memory cell contains a pointer to the list of instructions which are waiting for that cell as input. In our case, Mx, My, M1, and M2 are all needed by exactly one instruction each, and Mw is needed by two instructions, which are linked together by the "link" field of operand 3 of instruction I9.
4. In each instruction, ready bits are turned on for all output operands, and for all input operands which are currently ready (in our case, Ma and Mb).
5. Those instructions which have all their ready bits on are placed in the Ready List. In our case, this includes instructions I7, I8, and I10.

During the second phase of execution, the instructions in the Ready List are executed. This makes ready the operands of pther instructions, which are executed in turn until all processing is complete. In our example, the execution of I7, I8, and I10 will make ready memory cells Mw, M1, and M2. The processors will then execute ISRB storage cycles on the instructions pointed to by the link fields of these memory cells. As a result, instructions I9

and I11 will be seen to be ready, and will be placed on the Ready List. When these instructions have been executed, memory cells Mx and My will be made ready, which in turn will trigger the final instruction, I12. The state of the system after processing is complete is shown in Figure 5.2.



READY LIST

(I7)	-	Mw,Ma,Mb
(I8)	+	M1,Ma,Mb
(I10)	*	M2,Ma,Mb

FIGURE 5.1

EXAMPLE PROGRAM AFTER EXPANSION

MEMORY

	Value	Link
(Ma)	A	
(Mb)	B	
(Mw)	A-B	
(Mx)	(A+B)/(A-B)	
(My)	(A*B)/(A-B)	
(Mz)	T or F	
(M1)	A+B	
(M2)	A*B	

INSTRUCTION STORE

	Opcode	Operands (R)=ready bit on	Stat. Link
(I7)	-	Mw(R),Ma(R),Mb(R)	
(I8)	+	M1(R),Ma(R),Mb(R)	
(I9)	/	Mx(R),M1(R),Mw(R)	
(I10)	*	M2(R),Ma(R),Mb(R)	
(I11)	/	My(R),M2(R),Mw(R)	
(I12)	>	Mz(R),Mx(R),My(R)	

READY LIST

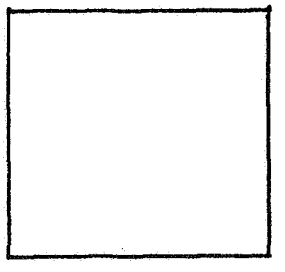


FIGURE 5.2

EXAMPLE PROGRAM AFTER EXECUTION IS COMPLETE

CHAPTER SIX

EVALUATION AND CONCLUSIONS

SIMULATION EXPERIMENTS

This chapter will describe a series of simulation experiments intended to investigate the behavior of the proposed system under various conditions, and to measure the speed advantage which might be realized by single-assignment parallel processing as compared with conventional, serial processing. The simulations to be described in this chapter were done by hand.

For the purpose of the experiments, an example problem has been chosen, and programmed both in SAMPLE and in IBM System/360 Assembler Language (23)(25), using those algorithms which best exploit the advantages of the respective systems. The problem chosen was multiplication of two square matrices. The SAMPLE program and the 360 program were both written to handle square matrices of any size without changing the code of the program. In each case, the operand matrices are assumed to be present in memory at the start of processing, and the product matrix is left in memory when processing is concluded. The 360 program, which is listed in Appendix C, was written in such

a way as to minimize memory accesses by using registers for storage of intermediate results. The SAMPLE program, which is listed in Appendix D, is based on the algorithm used as a programming example in Chapter 2. Although our measurements will be directly applicable only to the matrix multiplication program, it is hoped that the behavior of this program is representative of a larger class of numeric problems which might be programmed on the SAMPLE system.

In simulating the behaviors of the SAMPLE and 360 systems, some common measure of execution time is needed. The unit of measure selected is the memory cycle time. In the case of the 360, a "cycle" is considered to be a single memory access used to fetch an instruction, load a register, or store the contents of a register. In the case of the SAMPLE system, a "cycle" is a period of time in which each of the system's storage banks can service a single storage command, of one of the types listed in Chapter 3. By measuring the number of cycles used in the execution of the respective programs, we hope to measure the speed gain realized by the simultaneous activity of the various SAMPLE storage banks, as compared to the strictly serial memory accesses in the System/360 organization.

The ability of the SAMPLE system to exploit the opportunities for parallelism in its program is limited by the following four parameters of the system:

1. The number of processors
2. The number of ports per processor
3. The number of banks in the memory
4. The number of banks in the instruction store

A "port" is a means by which a processor issues storage commands to storage banks. If a processor has N ports, it may issue up to N storage commands in the same cycle; if these commands involve access to different storage banks, they may all be satisfied simultaneously. The memory addresses are interleaved among the banks in such a way that the bank in which a given address falls is equal to the address taken modulo the number of memory banks; the same rule holds for interleaving of instruction addresses among the instruction banks.

The System/360 organization will be compared with the SAMPLE organization in two ways: (1) comparison of the number of cycles required to complete their respective programs, subject to various constraints, and (2) comparison of storage requirements for instructions and data. In order to compensate for differences in word lengths between the two systems, all storage requirements will be measured in bits. In computing the storage requirements of the SAMPLE system, all addresses of instructions or memory cells are considered to be 20 bits long, giving the SAMPLE system an address space of 1,048,576 words.

The execution times and storage requirements of the System/360 for multiplying 2 x 2 and 3 x 3 matrices are summarized in Table 6.1.

The execution times and storage requirements of the SAMPLE system for the same problems, under certain constraints, are shown in Table 6.2. For the columns labelled "unlimited system", the SAMPLE system is considered to have as many processors, ports, and storage banks as can be utilized by the problem. The only constraint in such a system is that each individual instruction or memory cell can be accessed by only one processor in each cycle. Table 6.2 shows, for the unlimited system, the number of processors, ports, and storage banks which are needed in order to realize the theoretical minimum execution time for the given problem. However, this theoretical unlimited system is seen to require an unrealistic number of resources and to use them quite inefficiently. It would not be realistic, for example, to provide 102 access ports per processor, as required by the unlimited system for 3 x 3 matrices. So the same programs have been run on a limited system having exactly ten processors, four ports per processor, ten memory banks, and ten instruction store banks. Table 6.2 shows that the limited system has measured execution times only somewhat slower than the theoretical minimum for the given programs.

The results of Tables 6.1 and 6.2 are plotted in Figures 6.1 and 6.2. The following features of these figures should be noted:

1. The execution times of the SAMPLE system are both faster than the 360 and less sensitive to the size of the problem. The theoretical maximum speed factor realized by the unlimited SAMPLE system over the 360 is 3.0 for the 2 x 2 matrix multiplication and 7.9 for the 3 x 3 case.
2. The storage requirements of the 360 are both smaller than those of the SAMPLE system and less sensitive to the size of the problem. The storage requirements of the SAMPLE system exceed those of the 360 by a factor of 26.4 for the 2 x 2 case and 36.4 for the 3 x 3 case.

Further insight into the behavior of the two systems can be gained by studying the actual storage commands issued during processing. The 360 spends all its memory cycles on the conventional functions of fetching instructions and operands and storing results. These functions, which might be lumped under the heading "processing", are also performed by the SAMPLE system. However, the SAMPLE system must also perform two other functions not needed by the 360 organization. The first such function, called "expansion overhead", involves making multiple copies of instructions

in order to process iterated statements in the source program. Since the number of copies to be made is not known until run time, the compiler only provides a template for each instruction, which must be expanded into multiple instructions by means of the EXPAND instruction. The second unconventional function of the SAMPLE system, called "readiness overhead", involves searching for ready instructions. Whenever a memory cell becomes defined, the list of instructions which are waiting for that cell must be updated with the fact that the cell is ready, and any ready instructions must be placed on the Ready List.

It happens that the storage commands of the SAMPLE system are clearly differentiated into categories which fulfill the three functions of processing, expansion overhead, and readiness overhead. Table 6.3 shows the three categories of commands, and the numbers of commands of each category executed during processing of the 2 x 2 and 3 x 3 matrix multiplications. The bar graph in Figure 6.3 shows the relative magnitudes of these categories, along with the total number of cycles required by the 360 to solve the same problems. The following features of Figure 6.3 should be noted:

1. The SAMPLE system requires more storage cycles than the 360 for its "processing" function alone. This is probably due to the lack of internal registers for storage of intermediate results in the SAMPLE system.

2. The SAMPLE system spends approximately half its storage cycles on "non-processing" overhead functions.

Since the SAMPLE system organization requires more storage accesses to process a given problem than the 360 organization, it is clear that the SAMPLE system derives its speed advantage by overlapping accesses to different banks into the same cycle. Some feeling for how this occurs can be gained by examining the unlimited SAMPLE runs described previously. In these runs, the degree of overlap is limited only by the nature of the program and by the fact that each storage cell can only service one access request in a given cycle. Table 6.4 shows, for the unlimited 2 x 2 and 3 x 3 runs, the number of processors which issued storage access requests during each storage cycle in the history of the run. This data is plotted in Figure 6.4. Figure 6.4 reveals that the run contains two phases of approximately equal duration, which might be called the "expansion phase" and the "processing phase". During the expansion phase, relatively few processors are active, expanding instruction templates into multiple copies to allow for iteration. During the processing phase, the new instructions generated in the expansion phase are executed, performing the actual data manipulation of the program. Most of the opportunities for parallelism occur during the processing phase.

We have measured the execution time of the SAMPLE system with unlimited resources, and of the same system limited to ten processors, four ports per processor, ten memory banks, and ten instruction store banks. We will now consider each of these four types of resources individually, and investigate more fully the effect on the SAMPLE system of limiting each resource.

For each type of resource, we conduct a series of experiments in which the given resource is more and more limited, but all other resources are unlimited. The extreme cases for each series of experiments are the case in which the system is limited to only one of the given resource (one processor, memory bank, etc.), and the case in which the given resource, as well as all other resources, is unlimited. Execution times for the 2 x 2 matrix multiplication program are measured for these extreme cases, as well as intermediate cases, for each resource. The results are shown in Tables and Figures 6.5, 6.6, 6.7, and 6.8. The Figures show that when any resource is scarce, small changes in its availability make large changes in the execution time; however, when a resource is plentiful, the system is not sensitive to its availability. The curves also show that nearly optimum speed can be realized with relatively few copies of each resource; this is consistent with our previously observed result that the system with only ten processors, four ports per processor, ten memory banks, and ten instruction store banks performed only

somewhat more slowly than the unlimited system for this particular program. The ratio of execution times in the two extreme cases (only one resource, unlimited resources) is shown below for each resource:

processors:	ratio = 6.5
ports per processor:	ratio = 2.8
memory banks:	ratio = 5.8
instruction store banks:	ratio = 2.9

CONCLUSIONS

On the basis of the above experiments, the following conclusions may be drawn:

1. For certain problems, the SAMPLE scheme of single-assignment processing has speed advantages over conventional, serial processing. The SAMPLE system requires more memory accesses than a serial processor to accomplish the same unit of work, but the SAMPLE system is able to overlap these accesses, resulting in a net speed gain.
2. In order to realize its speed advantage, the SAMPLE system requires an expensive multiplicity of processors, storage banks, and storage access ports. In addition, the number of bits of storage required for a given unit of work is much greater in the SAMPLE system than in a conventional system.

3. The storage and other requirements of the SAMPLE system exceed those of a conventional system by a greater factor than its speed exceeds the speed of a conventional system. Therefore, the SAMPLE type of single-assignment processing is not a cost-effective way of utilizing a given quantity of hardware. Such a method of processing should be considered only in applications where raw speed is the only consideration, or as a research tool for investigating the properties of parallel systems.

SUGGESTIONS FOR CONTINUED RESEARCH

A number of areas suggest themselves for continued investigation and optimization of the SAMPLE system.

One such area is development of a parallel parsing technique, as suggested in the COMPILATION chapter. Particular attention might be given to how multiple parsing processors might interact to coordinate their activities in parsing a single program.

A second area for continued research concerns the scheduling of instructions in the case when the Ready List contains several ready instructions at the same time. As described in the "System" chapter, each Ready List bank treats its instructions in first-in, first-out fashion. However, other methods of scheduling the ready instructions could be studied. For example, a priority ordering could

be defined among the opcodes, such that certain opcodes would always be given precedence in the allocation of idle processors. Research could be conducted into choosing the best priority ordering among the opcodes, or into discovering an entirely different means of scheduling ready instructions.

A third possible research area is the problem of garbage collection. The size of the Memory and the Instruction Store in the proposed system tend to grow rapidly with the complexity of the problem being run. No algorithm has yet been proposed for de-allocation of an instruction cell which has already been executed, or a memory cell which is no longer needed. These problems are difficult for several reasons. Although an instruction has already been executed, its cell may still contain link fields which are active and which may still be used in updating other instructions. Similarly, although all instructions which use a given memory cell have already been executed, more instructions may be generated later which refer to the same cell, and so de-allocation of the memory cell is dangerous. In spite of these difficulties, it would seem that the resource de-allocation problem must be solved if the SAMPLE system is to be feasible.

TABLE 6.1

BEHAVIOR OF IBM SYSTEM/360 PROGRAM

	2 x 2 matrices	3 x 3 matrices
Execution Time (cycles)	164	484
Data Storage (bits)	416	896
Instruction Storage (bits)	576	576
Total Storage (bits)	992	1472

TABLE 6.2

BEHAVIOR OF SAMPLE PROGRAM

	2 x 2 matrices unlimited system	2 x 2 matrices limited system	3 x 3 matrices unlimited system	3 x 3 matrices limited system
Execution Time (cycles)	55	90	61	155
Processors	18	10	41	10
Ports per Processor	40	4	102	4
Memory Banks	95	10	195	10
Instruction Store Banks	64	10	128	10
Data Storage (bits)	7980	7980	16380	16380
Instruction Storage (bits)	18228	18228	35672	35672
Total Storage (bits)	26208	26208	52052	52052

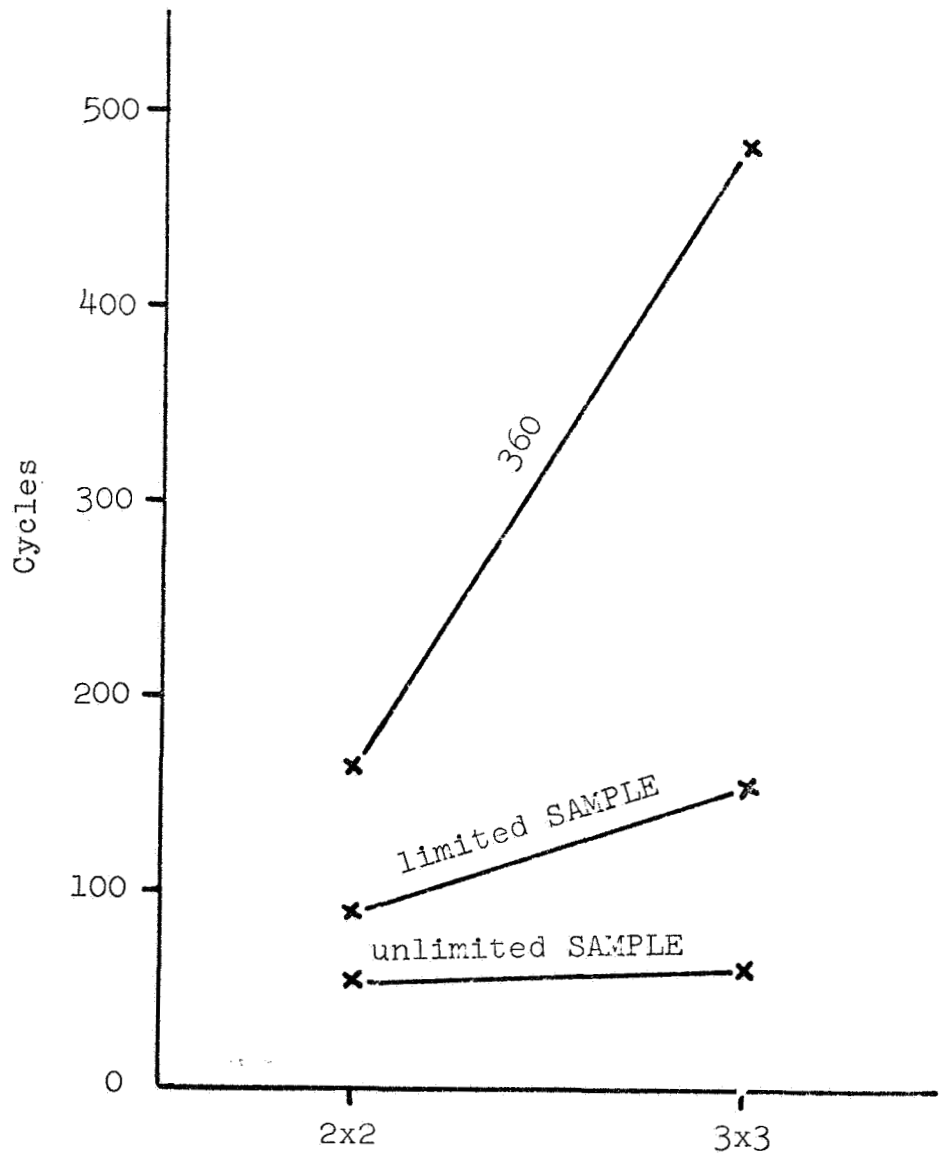


FIG. 6.1

EXECUTION TIME COMPARISON, 360 VS. SAMPLE

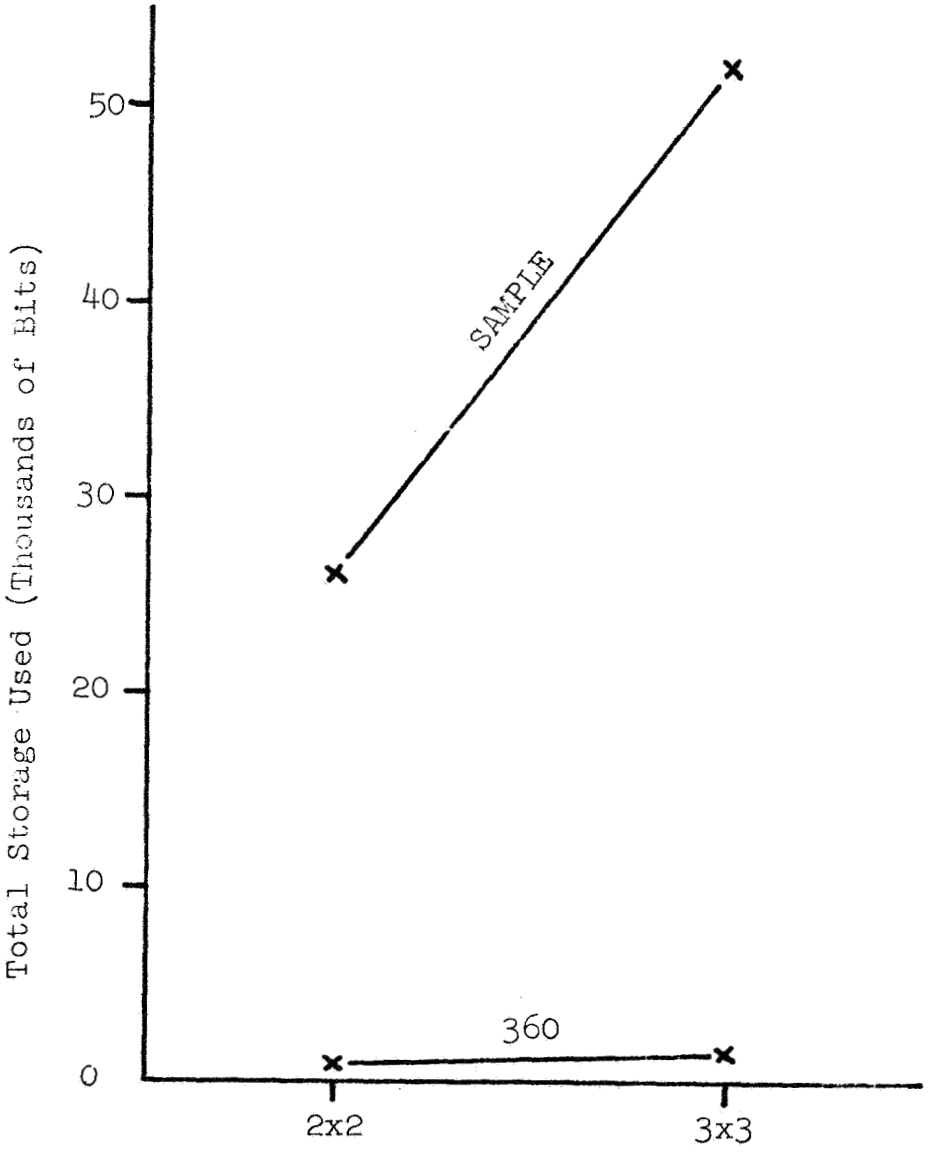


FIG. 6.2

STORAGE USAGE COMPARISON, 360 VS. SAMPLE

TABLE 6.3

FREQUENCY OF USE OF STORAGE COMMANDS

	2 x 2 matrices	3 x 3 matrices
Processing (total)	298	659
MR	175	400
MRW	175	400
MA	15	29
RF	40	76
Expansion Overhead (total)	149	301
MRWLC	72	143
IR	22	22
IW	48	121
IA	7	15
Readiness Overhead (total)	110	275
ISRB	75	203
RA	35	72
Grand Total	557	1235

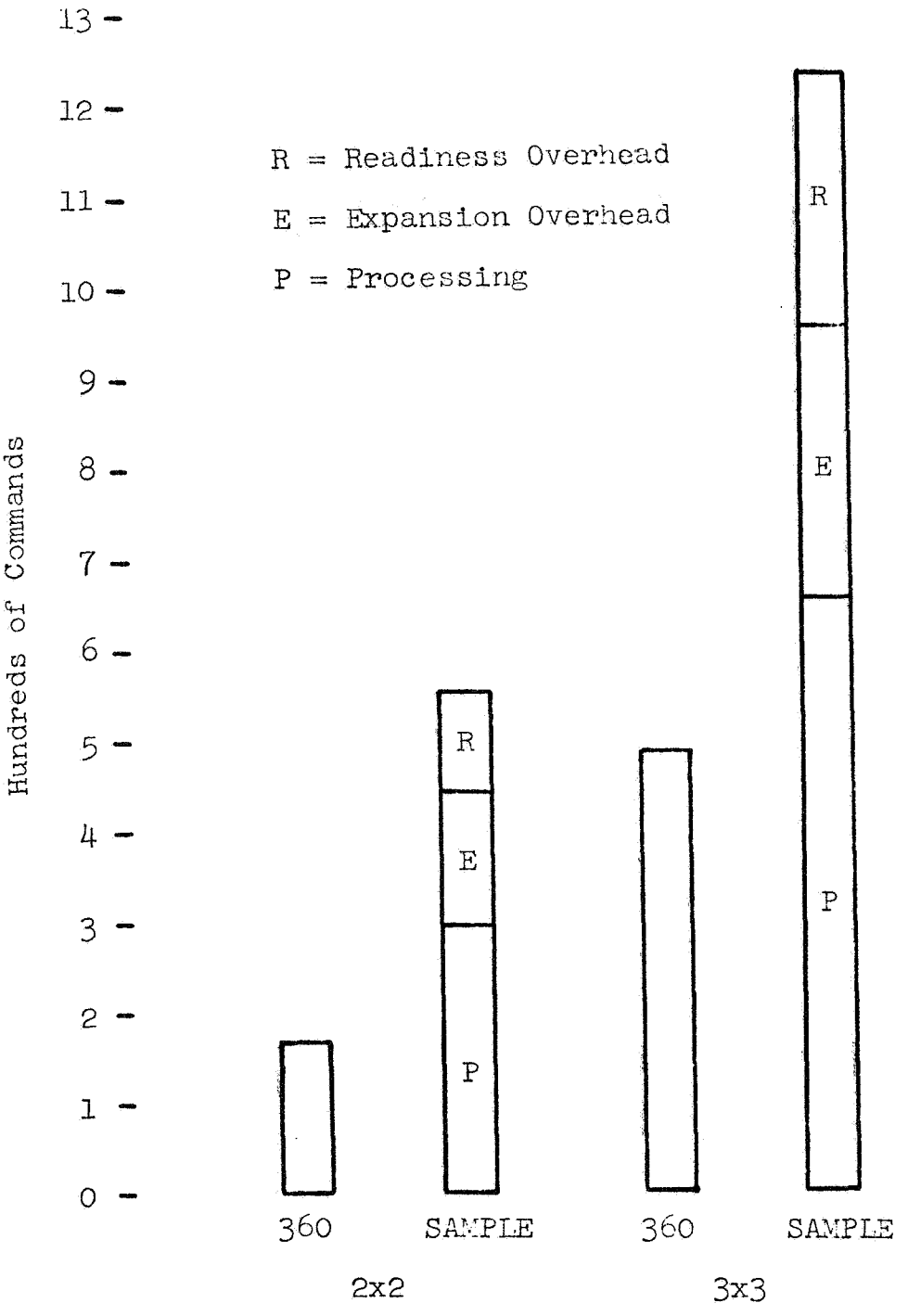


FIG. 6.3

USAGE OF STORAGE COMMANDS (BY TYPE)

TABLE 6.4

ACTIVE PROCESSORS VS TIME, UNLIMITED SAMPLE SYSTEM

Cycle	2 x 2 case	3 x 3 case	Cycle	2 x 2 case	3 x 3 case	Cycle	2 x 2 case	3 x 3 case
1	4	4	21	2	2	41	16	36
2	4	4	22	2	2	42	13	37
3	4	4	23	2	2	43	12	38
4	4	4	24	2	2	44	12	40
5	4	4	25	4	5	45	12	41
6	4	4	26	4	5	46	11	39
7	4	4	27	4	5	47	9	38
8	4	4	28	4	5	48	7	32
9	4	4	29	14	23	49	5	28
10	4	4	30	14	23	50	4	24
11	4	4	31	14	24	51	4	21
12	2	2	32	14	24	52	3	18
13	2	2	33	14	25	53	3	15
15	2	2	35	18	29	55	1	10
16	2	2	36	18	32	56		9
17	2	2	37	18	35	57		8
18	2	2	38	18	36	58		6
19	2	2	39	17	36	59		4
20	2	2	40	17	36	60		2
						61		1

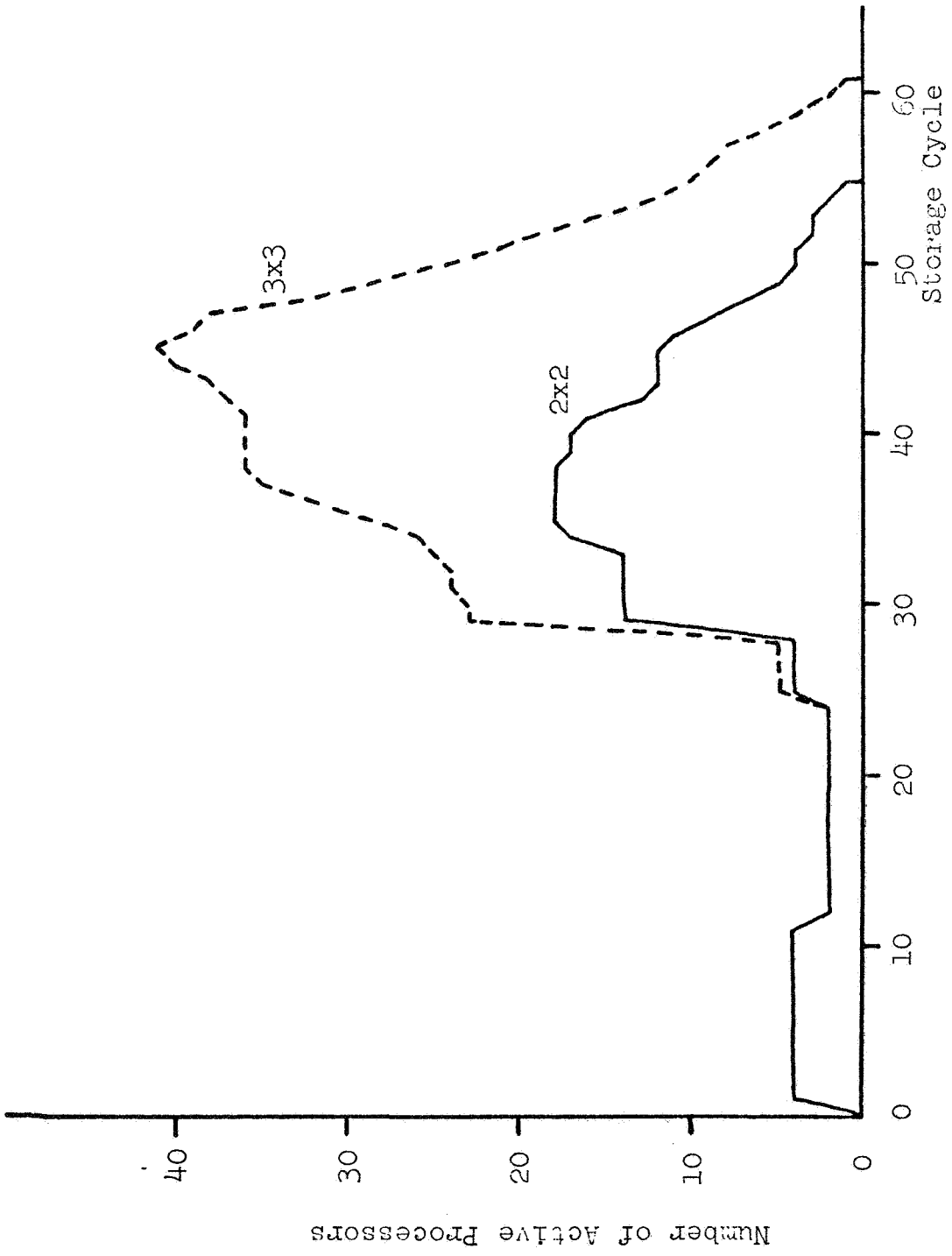


FIG. 6.4

ACTIVE PROCESSORS VS. TIME, UNLIMITED SAMPLE SYSTEM

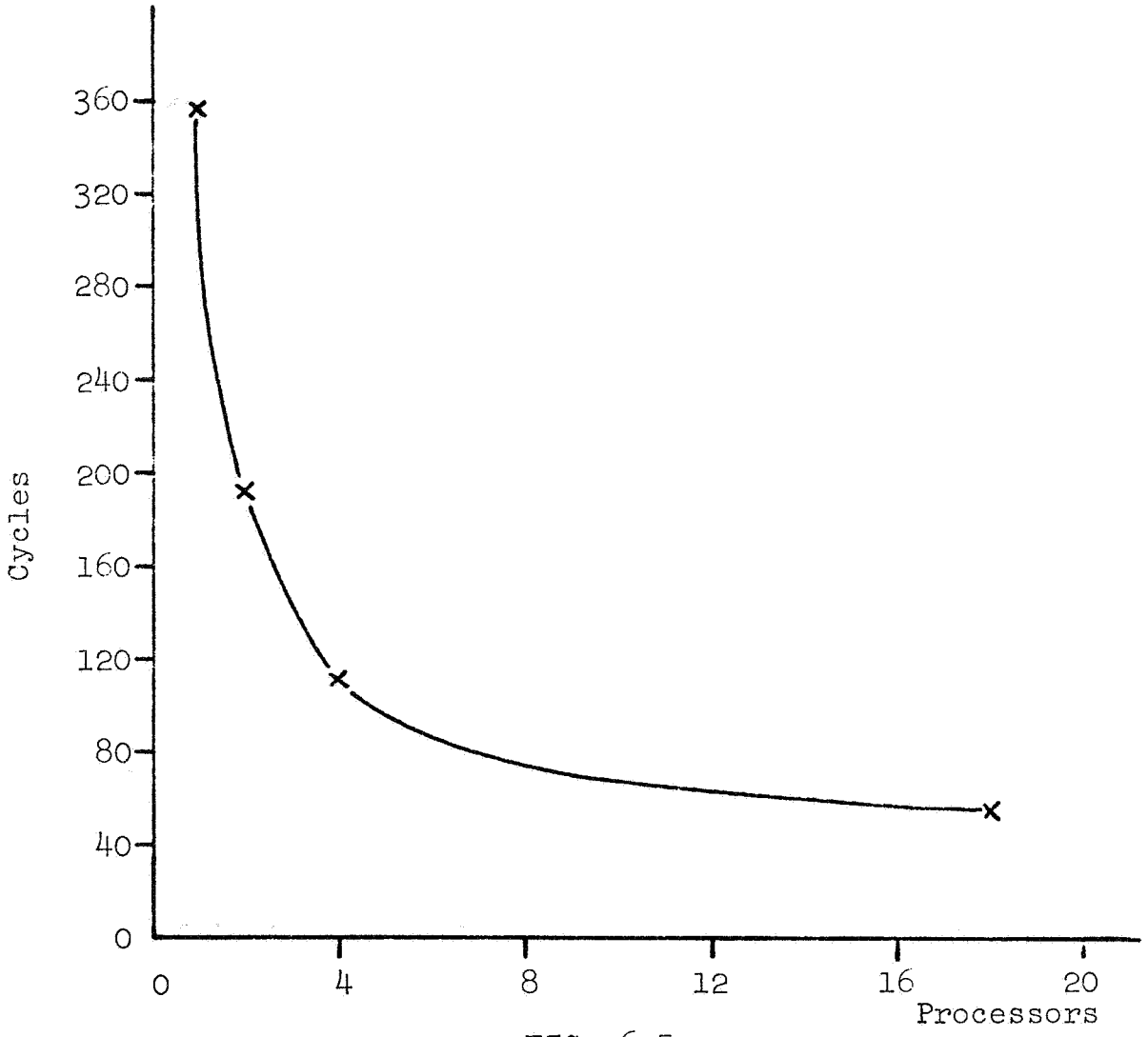


FIG. 6.5

Processors	Cycles
1	357
2	191
4	113
18	55

TABLE 6.5

SAMPLE EXECUTION TIME VS. PROCESSORS

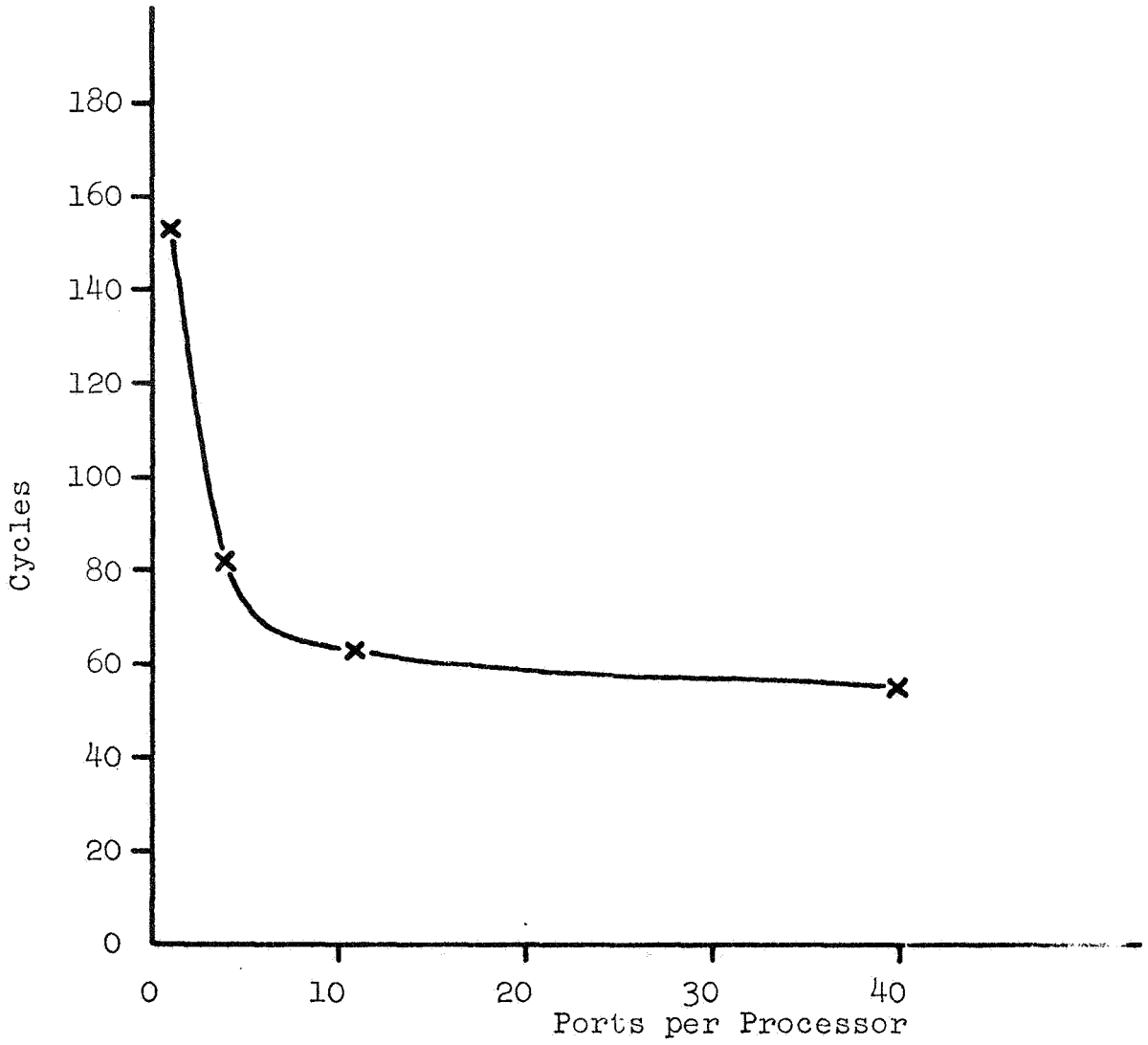


FIG. 6.6

Ports per Processor	Cycles
1	153
4	82
11	63
40	55

TABLE 6.6

SAMPLE EXECUTION TIME VS. PORTS PER PROCESSOR

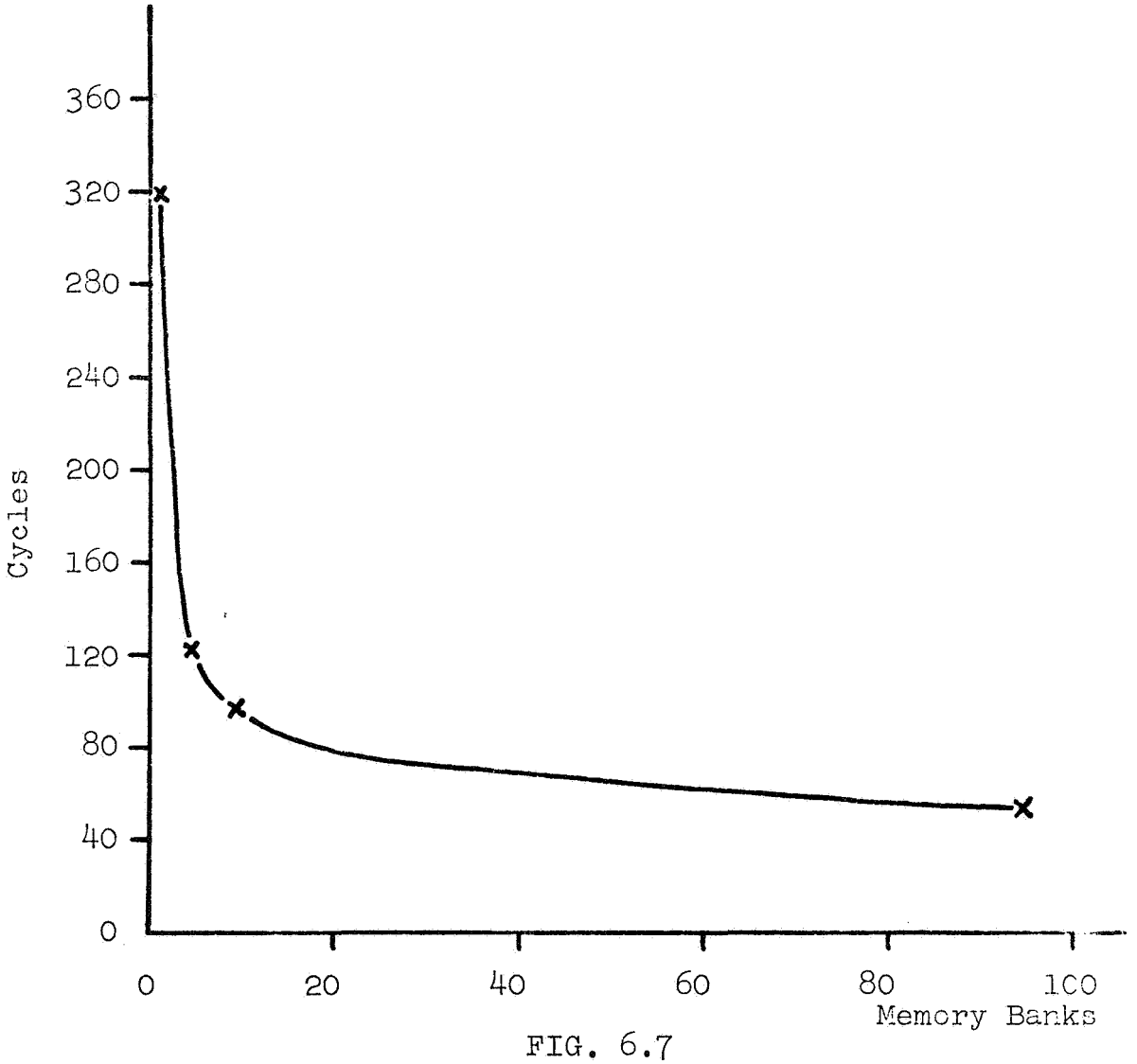


FIG. 6.7

Memory Banks	Cycles
1	320
5	123
10	98
95	55

TABLE 6.7

SAMPLE EXECUTION TIME VS. MEMORY BANKS

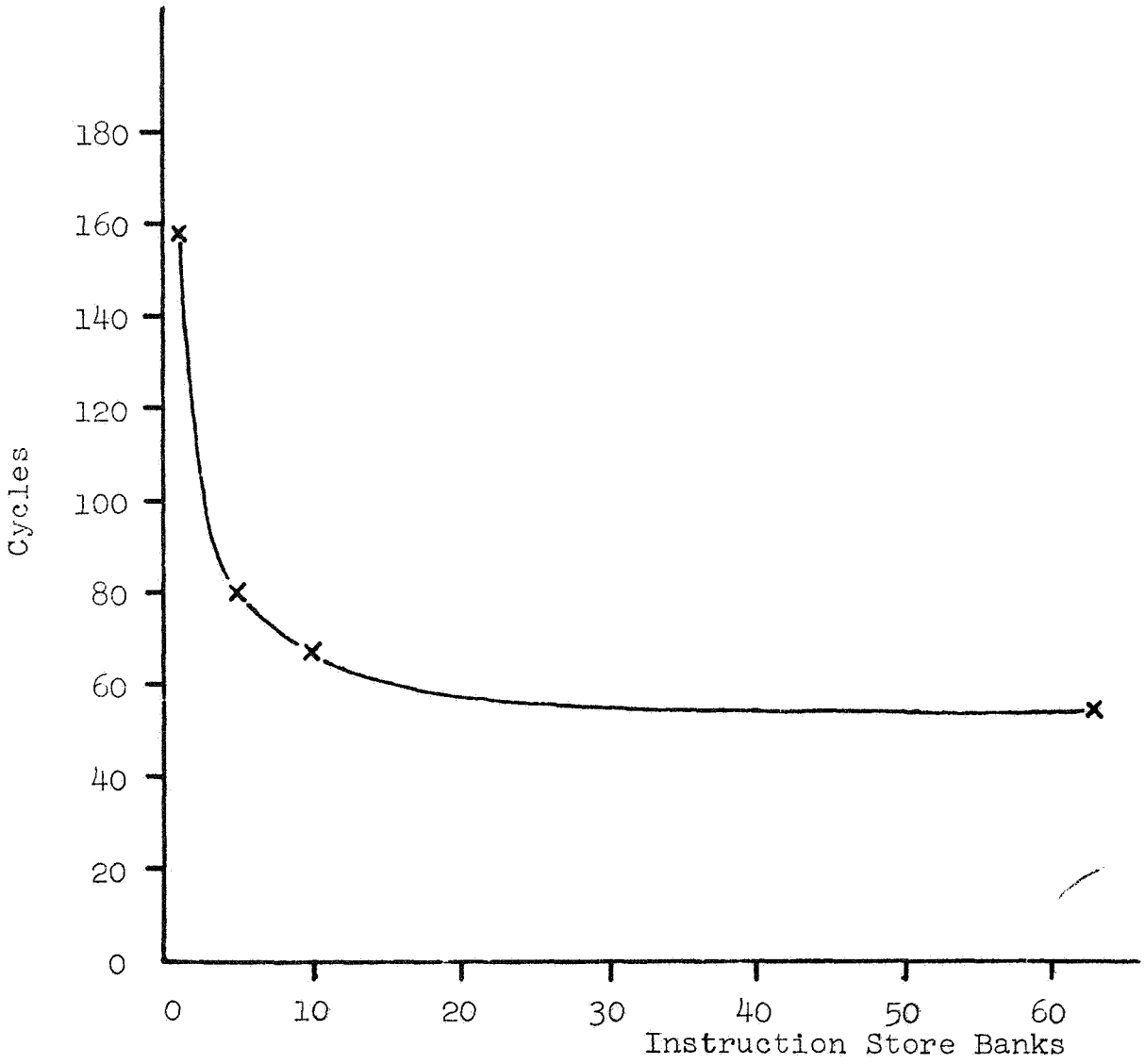


FIG. 6.8

Instruction Store Banks	Cycles
1	158
5	80
10	67
64	55

TABLE 6.8

SAMPLE EXECUTION TIME VS. INSTRUCTION STORE BANKS

APPENDIX A

SAMPLE SEMANTICS

The reductions of the SAMPLE grammar are listed, together with the actions to be taken by the compiler when each reduction is performed. In every reduction, it is implicit that the symbols on the right side will be replaced in the TSU by the symbol on the left. In the listing of a reduction, terminal symbols are represented by themselves, such as + . Nonterminal symbols are represented by their "type", "value", and "code begin" fields enclosed in brackets, such as <term.T1.I2> . Any of these fields may be omitted if its value is immaterial, or if it is to be treated by implicit conventions to be described later. The references in the "value" fields of the TSU cells must be of the type specified in the reduction listing, except that an M-reference may be substituted for a T-reference. In all symbols,

Mn means a memory address.

Tn means a tag without memory allocation.

In means a instruction address in the IS.

Ln means a literal integer value.

Any reference on the left side of a reduction which matches a symbol on the right side denotes that the reference is copied from the old symbol into the new symbol;

if a reference on the left side matches no reference on the right side, it is newly created. For example,

<term.T1> ::= <factor.T1>

denotes that the reference T1 is copied from the old symbol into the new symbol, whereas

<term.T1> ::= <term.T2> / <factor.T3>

denotes that the reference T1 is newly created. Each T-reference created by the parser is unique and distinct from all other T-references.

In the listings of reductions, the term "normal instruction chaining" has the following meaning:

1. The linked lists of instructions pointed to by the "code begin" fields of the symbols on the right side of the reduction are to be linked together into a single list. This can be accomplished by changing the "statement link" field of the instruction cell pointed to by the "code end" field of each fragment to point to the beginning cell of the next fragment.
2. Any new instructions emitted as a consequence of the reduction are added to the linked list at the head.
3. The new left side symbol in the TSU has its "code begin" and "code end" fields set to point to the

beginning and end of the new list of instruction cells.

The purpose of this convention is to ensure that each symbol in the TSU points to a linked list of instructions in the IS which are associated with the symbol. If the symbol has a value, the instruction which assigns its value will be the first instruction of the linked list. When the phrase "normal instruction chaining" is used with a reduction, the "code begin" fields of its symbols will not be explicitly shown in the listings below.

Some of the reductions call for emitting instructions with more than four operands. These instructions are placed in two consecutive cells in the IS, with the opcode of the second cell set to a special "continuation" code.

In some cases, it may be necessary to make an instruction wait for a dummy operand to become ready before the instruction is executed, although the dummy operand does not participate directly in the instruction. Dummy operands are denoted for any instruction by the use of parentheses. For example, ASSIGN M1,M2,(M3) behaves exactly like ASSIGN M1,M2 except that it cannot be performed until the dummy variable M3 is ready.

In the listings of reductions below, to "emit" an instruction means to place the instruction in a newly-allocated cell in the Instruction Store, but not to release it for execution. To "release an instruction for execution" results in the sequence of actions described in

detail under "Instruction Generate Routine" in the EXECUTION chapter.

<program> ::= <block.I1> .

Definition: I1 is the starting address of a linked list of one or more EXPAND or STARTLOOP instructions.

Action:

1. For every STARTLOOP instruction on the list I1, fill in literal 1 as its fifth operand, and the address of a newly-allocated memory cell as its fourth operand.
2. Release all instructions on the list I1 for execution on level 7.

<block.I1> ::= BEGIN <block head> <statement list.I1> END

No Action.

<block head> ::= <declaration>

No action.

<block head> ::= <block head> <function defn>

No Action.

<declaration> ::= DECLARE <name list> ;

This reduction is never seen by the parser because it is filtered out by the scanner.

<declaration> ::= DECLARE <name list>, FUNCTION <name list>;

This reduction is never seen by the parser because it is filtered out by the scanner.

<function defn> ::= DEFINE <name.I1> (<name list.I2>) ; <function body.I3> END

Definitions:

I1 is the start of a linked list of instruction cells containing T-references bound to variables declared inside the function, each labelled "internal" by a code in the opcode field.

I2 is the start of a linked list of instruction cells containing the parameters of the function, in order, each labelled "PARAMETER" by a code in the opcode field.

I3 is the start of a linked list of instruction cells, beginning with a T-reference labelled "RESULT", followed by one or more instructions, some of which may be EXPAND or STARTLOOP.

Action: Link lists I1, I2, and I3 together to form a single list beginning at I1.

<function defn> ::= DEFINE <name.I1> ; <function body.I2> END

Definitions:

I1 is the start of a linked list of instruction cells containing T-references bound to variables declared inside the function, each labelled

"internal" by a code in the opcode field. I2 is the start of a linked list of instruction cells, beginning with a T-reference labelled "RESULT", followed by one or more instructions, some of which may be EXPAND or STARTLOOP.

Action: Link lists I1 and I2 together to form a single list beginning at I1.

<name list> ::= <name>

This reduction is never seen by the parser because it is filtered out by the scanner.

<name list> ::= <name list> , <name>

This reduction is never seen by the parser because it is filtered out by the scanner.

<function body.I1> ::= <declaration> <statement list.I2> <expr.T3.I4>

Into a newly-allocated instruction cell I1, place the reference T2, with the label "RESULT" in the opcode field. Link the instruction lists beginning at I2 and I4 together to form a single linked list beginning with the new cell I1.

<function body.I1> ::= <statement list.I2> <expr.T3.I4>

Into a newly-allocated instruction cell I1, place the reference T2, with the label "RESULT"

in the opcode field. Link the instruction lists beginning at I2 and I4 together to form a single linked list beginning with the new cell I1.

<function body.I1> ::= <expr.T2.I3>

Into a newly-allocated instruction cell I1, place the reference T2, with the label "result" in the opcode field. Link onto the new cell I1 the instruction list beginning at I3, forming a single linked list.

<statement list.I1> ::= <statement.I1>

No Action.

<statement list.I1> ::= <statement list.I1> <statement.I2>

Link the list of instructions starting at I2 onto the end of the list of instructions starting at I1.

<statement.I1> ::= <block.I1>

No action.

<statement.I1> ::= <loop.I1>

No action.

<statement.I1> ::= READ <read list.I2> ;

1. Scan the list I2 and find all cells labelled "ITERANT".
2. Emit the following instruction (not on the list):

EXPAND I2

Set the "statement link" field of the new instruction to "none". Use all "ITERANT" T- or M-references found in step 1 as dummy operands of the EXPAND instruction (up to three operands).

3. Set the reference I1 in the new TSU symbol to the address of the new EXPAND instruction.

<statement.I1> ::= WRITE <write list.I2> ;

1. Scan the list I2 and find all cells labelled "ITERANT".
2. Emit the following instruction (not on the list):

EXPAND I2

Set the "statement link" field of the new instruction to "none". Use all "ITERANT" T- or M-references found in step 1 as dummy operands of the EXPAND instruction (up to three operands).

3. Set the reference I1 in the new TSU symbol to the address of the new EXPAND instruction.

<statement.I1> ::= <left part.T2.I3> IS TUPLE

(<num expr.T4.I5> , <num expr.T6.I7>) ;

1. Emit TUPLE T2,T4,T6
2. Link instruction lists I3, I5, and I7 (if any) together to form a single list starting at I3. Add the newly emitted TUPLE instructions to this list. Scan the list and find all cells labelled "ITERANT".

3. Emit the following instruction (not on the list):

   EXPAND I3

   Set the "statement link" field of the new instruction to "none". Use all "ITERANT" T- or M-references found in step 2 as dummy operands of the EXPAND instruction (up to three operands).

4. Set the reference I1 in the new TSU symbol to the address of the new EXPAND instruction.

<statement.I1> ::= <left part.T2.I3> ← <expr.T4.I5> ;

Definitions:

T2 is the tag or memory address to which the assignment is to be made.

I3 is the start of a list of instructions emitted during the parsing of <left part>.

T4 is the tag or memory address which is to be assigned.

I5 is the start of a list of instructions which define T4. The instruction which actually assigns a value to T4 is the first instruction on the list, and T4 is its first operand.

Action:

1. If the list I5 exists, change the first operand of instruction I5 from T4 to T2. Otherwise, emit the instruction ASSIGN T2,T4 at some location I5.

2. Link the lists I3 and I5 (if any) together to form a lingle list starting at I3. Scan this list and find all cells labelled "ITERANT".
3. Emit the following instruction (not on the list):

   EXPAND I3

   Set the "statement link" field of the new instruction to "none". Use all "ITERANT" T- or M-references found in step 2 as dummy operands of the EXPAND instruction (up to three operands).
4. Set the reference I1 in the new TSU symbol to the address of the new EXPAND instruction.

<read list.I1> ::= <read atom.I1>

No Action.

<read list.I1> ::= <read list.I1> , <read atom.I2>

Link the instruction list starting at I2 onto the end of the instruction list starting at I1.

<read atom.I1> ::= (<left part.T2.I3> , <num expr.T4.I5>)

1. Emit READ T2,T4
2. Link the instruction lists starting at I3 and I5 together with the new READ instruction to form a linked list starting at I1.

<write list.I1> ::= <write atom.I1>

No Action.

<write list.I1> ::= <write list.I1> , <write atom.I2>

Link the instruction list starting at I2 onto the end of the instruction list starting at I1.

<write atom.I1> ::= (<expr.T2.I3> , <num expr.T4.I5>)

1. Emit WRITE T2,T4
2. Link the instruction lists starting at I3 and I5 together with the new WRITE instruction to form a linked list starting at I1.

<left part.M1> ::= <name.M1>

No action.

<left part.T1> ::= <bounded left part.T1>

No action.

<left part.T1> ::= <unbounded left part.T1>

No action.

<bounded left part.T1> ::= <name.M2> <subscript.T3>
OF <num expr.T4>
Place the number 0 in a newly-allocated memory
cell Mp and mark it ready on level 7.
Emit TUPLE M2,Mp,T4
INSERT T1,M2,T3
Normal instruction chaining.

<bounded left part.T1> ::= <name.M2> <subscript.T3> OF
(<num expr.T4> , <num expr.T5>)
Emit TUPLE M2,T4,T5
INSERT T1,M2,T3
Normal instruction chaining.

<bounded left part.T1> ::= <bounded left part.T2>
<subscript.T3> OF <num expr.T4>
Place the number 0 in a newly-allocated memory
cell Mp and mark it ready on level 7.
Emit TUPLE T2,Mp,T4
INSERT T1,T2,T3
Normal instruction chaining.

<bounded left part.T1> ::= <bounded left part.T2>
<subscript.T3> OF (<num expr.T4> , <num expr.T5>)
Emit TUPLE T2,T4,T5
INSERT T1,T2,T3
Normal instruction chaining.

<unbounded left part.T1> ::= <leader.T2> <subscript.T3>

Emit INSERT T1,T2,T3

Normal instruction chaining.

<leader.M1> ::= <name.M1>

No action.

<leader.T1> ::= <leader.T2> <subscript.T3>

Emit SUB T1,T2,T3

Normal instruction chaining.

<loop.I1> ::= FOR <name.M2> ← <num expr.T3.I4> STEP <num expr.T5.I6> UNTIL <num expr.T7.I8> DO <init.I9.I10> ; <statement list.I11> END

Definitions:

I9 is the beginning of a list of instructions having no unallocated (T-reference) operands. The instructions initialize all INITIAL variables of the loop, and, when this is complete, define some memory cell Mp. Mp is the first operand of the first instruction on the list.

I10 is the beginning of a list of all M-references appearing in the initialization statement, in cells labelled "DOUBLE", and other M-references used in the initialization process, in cells labelled "STARTUP".

I11 is the beginning of a list of EXPAND and STARTLOOP instructions which point to all code for the loop.

Action:

1. Replace all T-references appearing on lists I4, I6, and I8 with newly-allocated M-references. (Note: if any of these lists do not exist, then the corresponding T-reference T3, T5, or T7 will instead be an M-reference, which we will refer to as M3, M5, or M7.)
2. Add all M-references newly allocated in step 1 to the list I10, in cells labelled "STARTUP".
3. Add the instruction ASSIGN M2,M3,(Mp) to the linked list at I9, where M3 is the M-reference substituted for T3 in step 1, and Mp is the first operand of the first instruction of the list I9.
4. Link the lists I4, I6, and I8 (if any) onto the end of the list I9.
5. Add the following instructions to the list I9:

   TEST Mr,M2,M7,M5

   UPDATE I11,I10,Mr,_,_,_,M2,M5,M7

   where Mr is newly allocated and M5 and M7 are the M-references substituted for T5 and T7.
6. Emit the following instruction (not on any list):

   STARTLOOP I11,I10,I9,_,_ .

   Place I1, the address of the new STARTLOOP

instruction, into the new TSU symbol.

7. Add M2 and Mr to the linked list of references at I10, in cells labelled "STARTUP".
8. For each STARTLOOP instruction on the list I11, place the second operand of the STARTLOOP instruction on the list I10, in a cell labelled "NESTED".
9. For each LEXPAND instruction on the list I11, scan the instructions on the list pointed to by the first operand of the LEXPAND instruction. Add to the list I10, in cells labelled "LOCAL", all M-references appearing as output operands of scanned instructions, except those already labelled "DOUBLE".

<loop.I1> ::= FOR <name.M2> ← <num expr.T3.I4>
UNTIL <num expr.T5.I6> DO <init.I7.I8> ;
<statement list.I9> END

Definitions:

I7 is the beginning of a list of instructions having no unallocated (T-reference) operands. The instructions initialize all INITIAL variables of the loop, and, when this is complete, define some memory cell Mp. Mp is the first operand of the first instruction on the list.

I8 is the beginning of a list of all M-references appearing in the initialization statement, in

cells labelled "DOUBLE", and other M-references used in the initialization process, in cells labelled "STARTUP".

I9 is the beginning of a list of EXPAND and STARTLOOP instructions which point to all code for the loop.

Action:

1. Replace all T-references appearing on lists I4 and I6 with newly-allocated M-references. (Note: if any of these lists do not exist, then the corresponding T-reference T3 or T5 will instead be an M-reference, which we will refer to as M3 or M5.)
2. Add all M-references newly allocated in step 1 to the list I8, in cells labelled "STARTUP".
3. Add the instruction ASSIGN M2,M3,(Mp) to the linked list at I7, where M3 is the M-reference substituted for T3 in step 1, and Mp is the first operand of the first instruction of the list I7.
4. Link the lists I4 and I6 (if any) onto the end of the list I7.
5. Add the following instructions to the list I7:

```
<=  Mr,M2,M5
UPDATE  I9,I8,Mr,_,_,_,M2,Mt,M5
```

where Mr is newly allocated, M5 is the M-reference substituted for T5, and Mt is

a memory cell which is newly allocated, set equal to 1, and marked ready on level 0.

6. Emit the following instruction (not on any list):

   STARTLOOP I9,I8,I7,_,_ .

   Place I1, the address of the new STARTLOOP instruction, into the new TSU symbol.

7. Add M2 and Mr to the linked list of references at I8, in cells labelled "STARTUP".

8. For each STARTLOOP instruction on the list I9, place the second operand of the STARTLOOP instruction on the list I8, in a cell labelled "NESTED".

9. For each LEXPAND instruction on the list I9, scan the instructions on the list pointed to by the first operand of the LEXPAND instruction. Add to the list I8, in cells labelled "LOCAL", all M-references appearing as output operands of scanned instructions, except those already labelled "DOUBLE".

<loop.I1> ::= WHILE <num expr.T2.I3> DO <init.I4.I5> ; <statement list.I6> END

Definitions:

I4 is the beginning of a list of instructions having no unallocated (T-reference) operands. The instructions initialize all INITIAL variables of the loop, and, when this is complete, define

some memory cell Mp. Mp is the first operand of the first instruction on the list.

I5 is the beginning of a list of all M-references appearing in the initialization statement, in cells labelled "DOUBLE", and other M-references used in the initialization process, in cells labelled "STARTUP".

I6 is the beginning of a list of EXPAND and STARTLOOP instructions which point to all code for the loop.

Action:

1. Link onto the end of the list starting at I4 a new copy of all the instructions on the linked list starting at I3. In the newly-copied instructions, replace all T-references by newly-allocated M-references. (Note: If the list I3 does not exist, the symbol <num expr> will contain some M-reference Ms: <num expr.Ms> In this case, allocate a new memory cell M2, and place in some location, which we will call I3, a new instruction ASSIGN M2,Ms .)
2. Add all M-references newly allocated in step 1 to the list I5, in cells labelled "STARTUP".
3. Add the following instructions to the linked list starting at I4:

```
ASSIGN  Mr,M2,(Mp)
REPEAT  I6,I5,Mr,_,_,I3
```

where Mr is newly allocated, M2 is the M-reference substituted for T2 (or the output operand of the newly-generated ASSIGN instruction) in step 1, and Mp is the first operand of the first instruction of the list I4.

4. Emit the following instruction (not on any list):

   STARTLOOP I6,I5,I4,_,_ .

   Place I1, the address of the new STARTLOOP instruction, into the new TSU symbol.

5. Add Mr to the linked list of references at I5, in a cell labelled "STARTUP".

6. For each STARTLOOP instruction on the list I6, place the second operand of the STARTLOOP instruction on the list I5, in a cell labelled "NESTED".

7. For each LEXPAND instruction on the list I6, scan the instructions on the list pointed to by the first operand of the LEXPAND instruction. Add to the list I5, in cells labelled "LOCAL", all M-references appearing as output operands of scanned instructions, except those already labelled "DOUBLE".

<init.I1.I2> ::= INITIAL <init atom.I1.I2>

No Action.

<init.I1.I2> ::= <init.I3.I4> , <init atom.I5.I6>

Definitions:

I5 is the start of a linked list of instructions which initialize some variable. The variable being initialized is the first operand of the first instruction of the list. The list contains no unallocated T-references.

I6 is the start of a linked list of all M-references used in initializing the variable, in cells labelled "startup", and the variable itself, in a cell labelled "double".

Action:

1. Emit NOP Mp,Mr,Ms where Mp is newly allocated, and Mr and Ms are the first operands of instructions I3 and I5, respectively.
2. Form the new NOP instruction, together with linked lists starting at I3 and I5, into a new linked list starting at cell I1, with the new NOP at the head of the list.
3. Link lists starting at I4 and I6 into a new linked list starting at I2. Add to this list Mp, the new M-reference allocated in step 1, in a cell with the label "startup" in the opcode field.

<init atom.I1.I2> ::= <name.M3> ← <expr.T4.I1>

1. Place the reference M3 in a cell I2, with the label "DOUBLE" in the opcode field.
2. Allocate a new memory cell and place its address in the "OLD" field of the cell M3.
3. In the instruction I1, change the first operand from T4 to M3. If the instruction I1 does not exist, emit the instruction ASSIGN M3,T4 in location I1.
4. Allocate a memory cell for each T-reference on the instruction list beginning at I1, and replace the T-references by the new M-references.
5. Add all the M-references newly allocated in step 4 to the list starting at I2, one per cell, with the label "startup" in the opcode field.

<expr.T1> ::= <num expr.T1>

No action.

<expr.T1> ::= <expr.T2> WITH <num expr.T3> ← <num expr.T4>

Emit WITH T1,T2,T3,T4

Normal instruction chaining.

<num expr.T1> ::= <logical expr.T1>

No action.

<num expr.T1> ::= IF <num expr.T2> THEN <num expr.T3> ELSE
<num expr.T4>
Emit IF T1,T2,T3,T4
Normal instruction chaining.

<logical expr.T1> ::= <logical term.T1>
No Action.

<logical expr.T1> ::= <logical expr.T2> OR <logical term.T3>
Emit OR T1,T2,T3
Normal instruction chaining.

<logical term.T1> ::= <logical factor.T1>
No Action.

<logical term.T1> ::= <logical term.T2> AND <logical factor.T3>
Emit AND T1,T2,T3
Normal instruction chaining.

<logical factor.T1> ::= <relation.T1>
No action.

<logical factor.T1> ::= NOT <relation.T2>
Emit NOT T1,T2
Normal instruction chaining.

<relation.T1> ::= <arith expr.T1>

No action.

<relation.T1> ::= <arith expr.T2> = <arith expr.T3>

Emit = T1,T2,T3

Normal instruction chaining.

<relation.T1> ::= <arith expr.T2> ¬= <arith expr.T3>

Emit ¬= T1,T2,T3

Normal instruction chaining.

<relation.T1> ::= <arith expr.T2> < <arith expr.T3>

Emit < T1,T2,T3

Normal instruction chaining.

<relation.T1> ::= <arith expr.T2> <= <arith expr.T3>

Emit <= T1,T2,T3

Normal instruction chaining.

<relation.T1> ::= <arith expr.T2> > <arith expr.T3>

Emit > T1,T2,T3

Normal instruction chaining.

<relation.T1> ::= <arith expr.T2> >= <arith expr.T3>

Emit >= T1,T2,T3

Normal instruction chaining.

<arith expr.T1> ::= <term.T1>

No Action.

<arith expr.T1> ::= - <term.T2>

Emit NEG T1,T2

Normal instruction chaining.

<arith expr.T1> ::= <arith expr.T2> + <term.T3>

Emit + T1,T2,T3

Normal instruction chaining.

<arith expr.T1> ::= <arith expr.T2> - <term.T3>

Emit - T1,T2,T3

Normal instruction chaining.

<term.T1> ::= <factor.T1>

No Action.

<term.T1> ::= <term.T2> * <factor.T3>

Emit * T1,T2,T3

Normal instruction chaining.

<term.T1> ::= <term.T2> / <factor.T3>

Emit / T1,T2,T3

Normal instruction chaining.

<term.T1> ::= <term.T2> MOD <factor.T3>

Emit MOD T1,T2,T3

Normal instruction chaining.

<factor.T1> ::= <quantity.T1>

No action.

<factor.T1> ::= FIRST <quantity.T2>

Emit FIRST T1,T2

Normal instruction chaining.

<factor.T1> ::= LAST <quantity.T2>

Emit LAST T1,T2

Normal instruction chaining.

<factor.T1> ::= ROUND <quantity.T2>

Emit ROUND T1,T2

Normal instruction chaining.

<factor.T1> ::= FLOOR <quantity.T2>

Emit FLOOR T1,T2

Normal instruction chaining.

<factor.T1> ::= CEIL <quantity.T2>

Emit CEIL T1,T2

Normal instruction chaining.

<factor.T1> ::= ABS <quantity.T2>

    Emit ABS T1,T2

    Normal instruction chaining.

<factor.T1> ::= + <quantity.T2>

    Emit SUM T1,T2

    Normal instruction chaining.

<factor.T1> ::= * <quantity.T2>

    Emit PROD T1,T2

    Normal instruction chaining.

<factor.T1> ::= AND <quantity.T2>

    Emit TAND T1,T2

    Normal instruction chaining.

<factor.T1> ::= OR <quantity.T2>

    Emit TOR T1,T2

    Normal instruction chaining.

<factor.M1> ::= <number.M1>

    No action.

<factor.T1> ::= <unlabelled tuple.T1>

    No action.

<quantity.T1> ::= <primary.T1>

No action.

<quantity.T1> ::= <tuple element.T1>

No action.

<primary.M1> ::= <name.M1>

No action.

<primary.M1.I2> ::= ' <name.M1> '

The address M1 is entered into an instruction cell and marked "ITERANT"; the address of this cell becomes I2.

<primary.T1> ::= OLD <name.M2>

Emit OLD T1,M2 .

Normal instruction chaining.

<primary.T1> ::= <function call.T1>

No action.

<primary.T1> ::= (<expr.T1>)

No action.

<primary.M1> ::= NIL

A memory cell M1 is allocated.
However, no value will ever be assigned to this cell,
so it will never become ready, and no instruction
depending on this cell will ever be executed.

<primary.M1> ::= TRUE

Allocate a new memory cell M1, set its value
to -1, and mark it ready on level 7.

<primary.M1> ::= FALSE

Allocate a new memory cell M1, set its value
to 0, and mark it ready on level 7.

<function call.T1> ::= <name.I2> (<simple tuple.T3>)

Emit CALL T1,I2,T3
Normal instruction chaining.

<tuple element.T1> ::= <primary.T2> <subscript.T3>

Emit SUB T1,T2,T3
Normal instruction chaining.

<tuple element.T1> ::= <tuple element.T2> <subscript.T3>

Emit SUB T1,T2,T3
Normal instruction chaining.

<subscript.T1> ::= | <primary.T1>

No Action.

<subscript.T1> ::= | <number.M2>

Emit ROUND T1,M2

Normal instruction chaining.

<unlabelled tuple.T1> ::= < <simple tuple.T1> >

No action.

<unlabelled tuple.T1> ::= < <tuple specifier.T1> >

No action.

<unlabelled tuple.M1> ::= < >

Place in a newly-allocated memory cell M1 the representation of a null tuple. Mark this cell ready on level 7.

<simple tuple.T1> ::= <expr.T2>

1. Allocate three new memory cells Mp, Mr, and Ms, place the number 0 in each of them, and mark them all ready on level 7.
2. Emit TUPLE T1,Mp,Mr
   INSERT T2,T1,Ms

   Normal instruction chaining.

<simple tuple.T1.I2> ::= <simple tuple.T1.I2> , <expr.T3.I4>

1. Instruction cell I2 contains an instruction

   TUPLE T1,Mp,Mr.

   Read the content of memory cell Mr. Increment its content by one. In addition, allocate a new memory cell Mt, place in it a constant equal to the incremented value of Mr, and mark it ready on level 7.

2. Emit INSERT T3,T1,Mt

   Normal instruction chaining.

<tuple specifier.T1> ::= <num expr.T2> TO <num expr.T3>

1. Place the number 1 in a newly-allocated memory cell Mp and mark it ready on level 7.

2. Emit TOBY T1,T2,T3,Mp

   Normal instruction chaining.

<tuple specifier.T1> ::= <num expr.T2> TO <num expr.T3>

BY <num expr.T4>

Emit TOBY T1,T2,T3,T4

Normal instruction chaining.

APPENDIX B

DESCRIPTIONS OF MACHINE INSTRUCTIONS

The 48 instructions are listed and described below. Before any instruction is released for execution, all T-reference operands will have been replaced by M-references. The reference types of the operands must match those in the instruction descriptions below.

If, during processing of any instruction, a processor detects an anomalous condition (wrong type operand, overflow, etc.), all output operands of the instruction are set to the special value UNDEFINED. If any instruction encounters UNDEFINED as an input operand, all its output operands are set to UNDEFINED.

In the descriptions below, "input cells" are memory cells which must be ready on the same level as the instruction, or on a lower level, before the instruction can be executed. "Output cells" are memory cells which the instruction makes to be ready on the same level as the instruction.

Occasionally an instruction will be written with some of its operands in parentheses. These operands are dummy operands, which do not participate in execution of the instruction, but which must be ready before the instruction can be executed. For example, the instruction

```
ASSIGN  M1,M2,(M3)
```

assigns the value of M2 to M1 as soon as both M2 and M3 are ready.

The instructions STARTLOOP, UPDATE, and REPEAT have more than four operands each. Hence, each of these instructions occupies two consecutive cells in the IS. To allocate space for such an instruction, a processor must allocate two cells at once. To read such an instruction, two Instruction Read (IR) requests are needed. However, these instructions have input operands only among their first four operands. Therefore, the readiness-testing portions of the Readiness Routine and the Instruction Generate Routine need only deal with the first cell of the double-cell instruction, as usual. The operand "link" fields and "statement link" field of the second cell are not used. Because the Ready LIST HAS room for eight operands per instruction, only a single Readylist Add (RA) or Readylist Fetch (RF) request is needed for a double-length instruction.

+	SUM	EXPAND
-	PROD	LEXPAND
*	TAND	CALL
/	TOR	STARTLOOP
MOD	ESUM	UPDATE
AND	EPROD	REPEAT
OR	ETAND	OLD
ROUND	ETOR	NOP
FLOOR	ASSIGN	STEP
CEIL	TOBY	TEST
NEG	INSERT	
ABS	TUPLE	
NOT	SUB	
=	FIRST	
¬=	LAST	
>	IF	
>=	WITH	
<	READ	
<=	WRITE	

INSTRUCTION DESCRIPTIONS

@ M1,M2,M3 where @ is one of +, -, *, /, MOD, AND, OR

Input cells: M2,M3; Output cell: M1

If M2 and M3 are both numbers, the number M2 @ M3 is placed in M1. Note that this result need not be an integer even if M2 and M3 are integers.

If M2 is a tuple and M3 is a number, M1 is made to be a tuple with "first" and "last" fields equal to those of M2, and "start" field pointing at a newly-allocated block of cells. A new set of instructions @ Mi,Mj,M3 is emitted and released for execution, where Mj = each element of M2, and Mi = the corresponding element of the newly-allocated block.

If M3 is a tuple and M2 is a number, M1 is made to be a tuple with "first" and "last" fields equal to those of M3, and "start" field pointing at a newly-allocated block of cells. A new set of instructions @ Mi,M2,Mj is emitted and released for execution, where Mj = each element of M3, and Mi = the corresponding element of the newly-allocated block.

If M2 and M3 are both tuples, M1 is made to be a tuple with "first" and "last" fields equal to those of M2 and M3, and "start" field pointing to a newly-allocated

block of cells. (If M2 and M3 do not have identical "first" and "last" fields, M1 becomes UNDEFINED.) A new set of instructions @ Mi,Mj,Mk is emitted and released for execution, where Mj and Mk are corresponding elements of M2 and M3, and Mi is the corresponding element of the newly-allocated block.

% M1,M2 where % is one of ROUND, FLOOR, CEIL, NEG, ABS, NOT
Input cell: M2; Output cell: M1

If M2 is a number, the number %(M2) is placed in M1.

If M2 is a tuple, M1 is made to be a tuple with "first" and "last" fields equal to those of M2, and "start" field pointing at a newly-allocated block of cells. A new set of instructions % Mi,Mj is emitted and released for execution, where Mj = each element of M2, and Mi = the corresponding element of the newly-allocated block.

? M1,M2,M3 where ? is one of =, ¬=, >, >=, <, <=
Input cells: M2,M3; Output cell: Possibly M1

If M2 and M3 are both numbers, the value of the relation M2 ? M3 (TRUE or FALSE) is placed in M1. (Logical values have type "number"; TRUE = -1 and FALSE = 0.)

If only one of M2 and M3 is a tuple, the value FALSE is placed in M1, unless ? is ¬= . In this case, TRUE is placed in M1.

If both M2 and M3 are tuples, their "first" and "last" fields must be identical, or else FALSE is placed in M1 (except if ? is ¬= , TRUE is placed in M1). If the fields match, a memory allocation request is made for a connected group of cells equal to the number of elements in M2. Then the following new instructions are emitted into the IS and released for execution:

? Mi,Mj,Mk where Mj and Mk are corresponding elements of tuples M2 and M3, and Mi is the corresponding cell in the newly-allocated block.

ETAND M1,Mp,Lq where Mp is the starting address of the newly-allocated block, and Lq is the number of cells in the block. (Exception: if ? is ¬=, emit ETOR M1,Mp,Lq.)

\# M1,M2 where # is one of SUM, PROD, TAND, TOR

Input cell: M2; Output cell: None.

If M2 is not a tuple, M1 becomes UNDEFINED. Otherwise, the instruction X M1,Mp,Lq is emitted and released for execution, where Mp = the "start" field of M2, Lq is a literal equal to the number of elements in M2, and X is ESUM, EPROD, ETAND, or ETOR, depending on whether # is SUM, PROD, TAND, or TOR.

$ M1,M2,L3 where $ is one of ESUM, EPROD, ETAND, ETOR

Input cell: M2; Output cell: possibly M1

M2 is the first of a connected group of L3 cells which are to be combined by the operator +, *, AND, or OR, leaving the result in M1.

If L3 is 0, 0 is placed in M1.

If L3 is 1, the content of M2 is placed in M1. If M2 is not ready, the instruction ASSIGN M1,M2 is emitted and released for execution.

If L3 is 2, the contents of M2 and M2+1 are read, combined by the operator +, *, AND, or OR (depending on the nature of $), and the result is placed in M1. If either of cells M2 or M2+1 is not ready, the instruction X M1,M2,M2+1 is emitted and released for execution, where X is +, *, AND, or OR, depending on the nature of $.

If L3 is 3 or more, two newly-allocated cells Mp and Mq are requested, and the following instructions are emitted and released for execution:

X M1,Mp,Mq where X is +, *, AND, or OR, depending on $

$ Mp,M2,FLOOR(L3/2)

$ Mq,M2+FLOOR(L3/2),L3-FLOOR(L3/2)

ASSIGN M1,M2

Input cell: M2; Output cell: M1

The value of M2 is placed in M1.

TOBY M1,M2,M3,M4

Input cells: M2,M3,M4; Output cells: M1 and all newly allocated cells.

M2, M3, and M4 must be numbers. A complete tuple is created in memory cell M1, beginning with the number in M2 and continuing to the number in M3 by increments of M4. The "first", "last", and "start" fields of M1 are filled in, and M1 is marked ready. Cells are allocated, defined and made ready for all elements of the tuple.

INSERT M1,M2,M3

Input cells: M1,M2,M3; Output cell: the newly-defined element.

M2 must be a tuple. M3 must be a number. M1 is entered as element M3 of the tuple M2. Note that element M3 need not be the M3th element of the tuple if the "first" field of M2 is not 1.

TUPLE M1,M2,M3

Input cells: M2,M3; Output cell: M1

M2 and M3 must be numbers. The cell M1 is examined; if this cell is already marked ready on the level of the TUPLE instruction, no action is taken. Otherwise, M1 is made to be a tuple. The value of M2 is placed in its "first" field, and the value of M3 is placed in its "last" field. An area of free memory containing (M3)-(M2)+1 cells is allocated, and the "start" field of M1 is set to point at the first cell of this area.

SUB M1,M2,M3

Input cells: M2,M3; Output cell: Possibly M1

M2 must be a tuple, and M3 must be a number. If element M3 of tuple M2 is ready, it is placed in M1. Otherwise, the instruction ASSIGN M1,Mp is emitted and released for execution, where Mp is the address of element M3 of tuple M2.

FIRST M1,M2

Input cell: M2; Output cell: M1

M2 must be a tuple. M1 is made to be a number, and set equal to the "first" field of M2.

LAST M1,M2

Input cell: M2; Output cell: M1

M2 must be a tuple. M1 is made to be a number, and set equal to the "last" field of M2.

IF M1,M2,M3,M4

Input cells: M2,M3,M4; Output cell: M1

M2 must be a number.

If M2 is true (negative), the value of M3 is placed in M1. If M2 is false (non-negative), the value of M4 is placed in M1.

WITH M1,M2,M3,M4

Input cells: M2,M3; Output cell: M1

M2 must be a tuple. M3 must be a number representing a valid subscript in the tuple M2. M1 is made to be a tuple with "first" and "last" fields equal to those of M2, and "start" field pointing to a newly-allocated block of cells. All the following new instructions are emitted and released for execution:

ASSIGN Mi,Mj where Mj = each element of tuple M2 except element M3, and Mi = the corresponding element of the newly-allocated block.

ASSIGN Mk,M4 where Mk = element M3 of the newly-allocated block.

READ M1,M2

Input cell: M2; Output cell: M1 and possibly other, newly-allocated cells.

M2 contains an integer value. The input medium is scanned, and the input quantity associated with this integer is obtained, and loaded into M1. If the input quantity is a tuple or a nested tuple, new memory cells are allocated and loaded with the tuple elements of the input quantity.

WRITE M1,M2

Input cells: M1, M2; Output cell: None.

The content of cell M1 is transmitted to the output medium, and made to be associated with the integer value in M2. If M1 is a tuple, all its element values to arbitrary levels of nesting are transmitted, with appropriate notations to indicate nesting. If some element or elements, M3 and M4, of the tuple structure of M1 are not ready, no values are transmitted to the output medium, but instead a new instruction WRITE M1,M2,(M3),(M4) is emitted, having up to two of the non-ready elements as dummy operands.

EXPAND I1

No input or output cells.

I1 is the address of the first of a linked list of instructions, linked together by their "statement link" fields. Among these instructions, not necessarily at the head of the list, may be some instructions having the special opcode "ITERANT" and an M-reference as first operand. These M-references will be referred to as iterants. Duplicate iterant references may occur on the list, and should be ignored. The following is done:

1. First, all iterant M-references are read. If any iterant is not ready, no action is taken except that a new instruction EXPAND I1,(Mx),(My),(Mz) is emitted and released for execution, having as dummy operands all the non-ready iterants (up to 3).
2. If all iterants are ready, the processor constructs an internal table of "spans" having an entry for every T-reference appearing in the linked list of instructions. For each T-reference, the iterants (if any) which that T-reference spans are listed. Any output cell of an instruction spans all iterants spanned by any input cell of that instruction. In addition, the first operand of an INSERT instruction spans all iterants spanned by the second and third operands of the INSERT instruction. All other

T-references span nothing. We will say that an instruction spans all iterants which are spanned by any of its operands.

3. Memory space is allocated to the T-references in the linked list. Each T-reference is allocated a number of cells equal to the product of the numbers of elements in all the iterant tuples which it spans. The T-reference is replaced in all instructions by this M-reference. If a T-reference spans nothing, it is allocated one cell.
4. Throughout the linked list of instructions, each iterant is replaced by an M-reference equal to its "start" field; that is, the address of its first element.
5. An arbitrary ordering is defined among all the iterants, which will be referred to in part 6.
6. Each instruction is replicated as many times as the product of the numbers of elements in all the iterant tuples which it spans. Assume a given instruction spans N iterants, and there are Mi elements in the ith iterant. Each new copy of the instruction is associated with a unique N-tuple denoting an element number from 0 to Mi-1 for each iterant spanned by the instruction. Let the N-tuple associated with a given copy be (K1,K2,...,Kn). Each operand in the given copy has its address modified as follows. Assume that the particular

operand in question spans only those iterants in a set S. Assign to each iterant in the set S a weight equal to the product of the numbers of elements in all lower-ordered iterants in the set S. Let the weight of the ith iterant be Wi. Then the address of the operand in question is increased by the sum over all i in S of KiWi. Operands which span nothing have their address copied without change.

7. All the instructions generated in step 6 are placed in the IS and released for execution on the same level as the original EXPAND instruction.

By means of the above steps, each instruction in the original linked list is expanded to multiple copies, and the operands of the newly-generated copies are adjusted to span the space over which iteration is to occur.

LEXPAND I1,M2

No input or output cells.

This instruction behaves exactly the same as EXPAND I1, except that, in addition to the other instructions emitted, it emits a tree of newly-allocated NOP instructions which define the cell M2 to be ready as soon as all the newly-created instructions have executed. Memory cells are allocated as needed for the operands of the NOP instructions. All the new NOP

instructions are released for execution on the same level as the LEXPAND.

CALL M1,I2,M3

Input cell: M3; Output cell: None.

Definitions:

M1 is the address of the cell in memory to which the value of the function is to be assigned.

I2 is the address of the function in the IS, which consists of a list of "PARAMETER" and "INTERNAL" T-references, a "RESULT" T-reference, and some instructions which may have some T-references as operands.

M3 is the address in memory of a tuple containing the actual parameter values of the function call.

Action:

1. All instructions on the list I2, except the "INTERNAL", "PARAMETER" and "RESULT" cells, are copied over to a new place in the IS. Also copied over are lists which represent first operands of EXPAND or LEXPAND instructions, or first, second, or third operands of STARTLOOP instructions on the list I2, or in the copied material to any levels of nesting. In all newly-copied instructions, operand addresses are adjusted as necessary to point to the

newly-copied lists. Also, the link fields of newly-copied instructions are adjusted as necessary to preserve the linked-list properties of the newly-copied material. The following changes are made throughout the newly-copied material:

a. Formal parameters are replaced by M-references to the corresponding element of the tuple M3.

b. Any references to the result variable are replaced by the M-reference M1.

c. A unique M-reference is newly allocated for each "INTERNAL" T-reference, and is substituted for this T-reference throughout the newly-copied instructions.

2. Each STARTLOOP instruction on the new copy of the list I2 (first level only) has a newly-allocated M-reference filled in as its fourth operand, and receives as a fifth operand a literal equal to one more than the level of the CALL instruction.

3. Every instruction in the new copy of the list I2 (first level only) is released for execution on the same level as the CALL instruction. Note that instructions which are pointed to by first-level instructions are copied over but not released for execution.

STARTLOOP I1,I2,I3,M4,L5

No input or output cells.

Definitions:

I1 is the beginning of a linked list of LEXPAND (or EXPAND) and STARTLOOP instructions which point to all code for the loop. The code contains M-references for all variables which have names, and T-references for all other, compiler-generated variables. Each LEXPAND instruction is missing its second operand, and each STARTLOOP instruction is missing its fourth and fifth operands.

I2 is the beginning of the update list for the loop. This list may contain any of the following types of references:

a. "DOUBLE": M-references which are INITIALized in the loop.

b. "LOCAL": M-references which were allocated to named variables and which are assigned values inside the loop, except those labelled "DOUBLE".

c. "STARTUP": M-references used in initializing loop variables, which must be marked not ready each time the loop is called.

d. "NESTED": I-references which point to the update lists of nested loops.

I3 is the beginning of a linked list of instructions having no unallocated (T-reference) operands. The instructions initialize the index variable (if any) and all variables in the INITIAL statement, then test the continuation condition and, if it passes, issue an UPDATE or REPEAT instruction for the loop.

M4 is a memory cell which is to be defined when all loop iterations are complete.

L5 is a literal equal to the lexic level of the loop.

Note: Since STARTLOOP has five operands, it occupies two consecutive cells in the IS.

Action:

1. Each STARTLOOP instruction appearing on list I1 has its fifth operand set to L5+1 (if it does not equal this already).
2. Change the opcode of all EXPAND instructions on the list I1 to LEXPAND. The second operand of these instructions will be filled in later.
3. All "DOUBLE" and "STARTLOOP" M-references appearing on list I2 are marked not ready on level L5.
4. A new copy is made of all instructions on the list I3, and all the newly copied instructions are released for execution on level L5. The UPDATE or REPEAT instruction on the list I3 has

M4 filled in as the fourth operand and L5 as the fifth operand of its new copy (not of the original).

UPDATE I1,I2,M3,M4,L5,M6,M7,M8

Input cell: M3; Output cell: possibly M4.

Definitions:

I1 is the beginning of a linked list of LEXPAND and STARTLOOP instructions which point to all code for the loop. The code contains M-references for all variables which have names, and T-references for all other, compiler-generated variables. Each LEXPAND instruction is missing its second operand, and each STARTLOOP instruction is missing its fourth operand.

I2 is the beginning of the update list for the loop. This list may contain any of the following types of references:

a. "DOUBLE": M-references which are INITIALized in the loop.

b. "LOCAL": M-references which were allocated to named variables and which are assigned values inside the loop, except those labelled "DOUBLE".

c. "STARTUP": M-references used in initializing loop variables, which must be

marked not ready each time the loop is called.

d. "NESTED": I-references which point to the update lists of nested loops.

M3 is a trigger cell which starts the UPDATE. If M3 becomes TRUE, the loop should be repeated. If M3 becomes FALSE, the loop should be terminated.

M4 is a memory cell which is to be defined when all loop iterations are complete.

L5 is a literal equal to the lexic level of the loop.

M6 is the index variable of the loop.

M7 is the loop increment.

M8 is the upper limit for the loop index variable.

Note: Since UPDATE has 8 operands, it occupies two consecutive cells in the IS.

Action:

1. If M3 is TRUE, the following is done:

   a. For every "DOUBLE" M-reference Mi in the list I2, the value in Mi is placed in the cell pointed to by the "old" field of Mi, and this newly-filled cell is marked ready on level L5.

   b. All "LOCAL" and "DOUBLE" M-references on the list I2, and on all nested update lists to arbitrary levels, are marked not ready on level L5.

c. A new copy is made of all STARTLOOP and LEXPAND instructions in the list I1. In addition, new copies are made of all linked lists appearing as first operands of these LEXPAND instructions, and the LEXPAND operand references are adjusted to point to the new lists. Each new copy of a LEXPAND instruction receives a newly-allocated cell Mj as its second operand. Each new copy of a STARTLOOP instruction receives a newly-allocated cell Mk as its fourth operand. All the new STARTLOOP and LEXPAND instructions are released for execution on level L5.

d. Let Mj be the set of newly-allocated second operands of newly-copied LEXPAND instructions. Let Mk be the set of newly-allocated fourth operands of newly-copied STARTLOOP instructions. Emit a tree of NOP instructions which defines some new cell Mu when all cells Mj and Mk are ready. Release all the NOP instructions for execution on level L5.

e. Emit all the following instructions and release them for execution on level L5:

```
STEP  M6,M7,Mu,Mv
TEST  Mw,M6,M8,Mv
```

UPDATE I1,I2,Mw,M4,L5,M6,M7,M8

where Mv and Mw are newly-allocated cells.

2. If M2 is FALSE, the following is done:
   a. All "DOUBLE" and "LOCAL" M-references on the update list I2 and on all nested update lists are marked ready on level L5-1.
   b. M4 is made TRUE and ready on level L5-1.

REPEAT I1,I2,M3,M4,L5,I6

Input cell: M3; Output cell: possibly M4.

Definitions:

I1 is the beginning of a linked list of LEXPAND and STARTLOOP instructions which point to all code for the loop. The code contains M-references for all variables which have names, and T-references for all other, compiler-generated variables. Each LEXPAND instruction is missing its second operand, and each STARTLOOP instruction is missing its fourth operand.

I2 is the beginning of the update list for the loop. This list may contain any of the following types of references:
   a. "DOUBLE": M-references which are INITIALized in the loop.
   b. "LOCAL": M-references which were allocated to named variables and which are assigned

values inside the loop, except those labelled "DOUBLE".

c. "STARTUP": M-references used in initializing loop variables, which must be marked not ready each time the loop is called.

d. "NESTED": I-references which point to the update lists of nested loops.

M3 is a trigger cell which starts the REPEAT. If M3 becomes TRUE, the loop should be repeated. If M3 becomes FALSE, the loop should be terminated.

M4 is a memory cell which is to be defined when all loop iterations are complete.

L5 is a literal equal to the lexic level of the loop.

I6 is the beginning of a linked list of instructions which evaluate the continuation condition and place the result in the T- or M-reference which is the first operand of the first instruction of the list. The instructions contain M-references for all variables which have names, and T-references for other, compiler-generated variables.

Note: Since REPEAT has six operands, it occupies two consecutive cells in the IS.

Action:

1. If M3 is TRUE, the following is done:
   a. For every "DOUBLE" M-reference Mi in the list I2, the value in Mi is placed in the cell pointed to by the "old" field of Mi, and this newly-filled cell is marked ready on level L5.
   b. All "LOCAL" and "DOUBLE" M-references on the list I2, and on all nested update lists to arbitrary levels, are marked not ready on level L5.
   c. A new copy is made of all STARTLOOP and LEXPAND instructions in the list I1. In addition, new copies are made of all linked lists appearing as first operands of these LEXPAND instructions, and the LEXPAND operand references are adjusted to point to the new lists. Each new copy of a LEXPAND instruction receives a newly-allocated cell Mj as its second operand. Each new copy of a STARTLOOP instruction receives a newly-allocated cell Mk as its fourth operand. All the new STARTLOOP and LEXPAND instructions are released for execution on level L5.
   d. Let Mj be the set of newly-allocated second operands of newly-copied LEXPAND

instructions. Let Mk be the set of newly-allocated fourth operands of newly-copied STARTLOOP instructions. Emit a tree of NOP instructions which defines some new cell Mu when all cells Mj and Mk are ready. Release all the NOP instructions for execution on level L5.

e. Make a new copy of all instructions on the list I6. Allocate a new M-reference for every T-reference appearing on the list I6 and replace each T-reference with its newly-allocated M-reference in the new copy. Release all the newly-copied instructions for execution on level L5.

f. Emit the following instructions and release them for execution on level L5:

```
ASSIGN  Mw,Mv,(Mu)
REPEAT  I1,I2,Mw,M4,L5,I6
```

where Mv is the M-reference allocated in step (e) for the first operand of the first instruction of the list I6, and Mw is a newly-allocated cell.

2. If M3 is FALSE, the following is done:

a. All "DOUBLE" and "LOCAL" M-references on the update list I2 and all nested update lists are marked ready on level L5-1.

b. M4 is made TRUE and ready on level L5-1.

OLD M1,M2

Input cells: none; Output cell: possibly M1

The cell M2 is examined to find the address Mp in its "OLD" field. If memory cell Mp is ready, its value is placed in M1. If not, the instruction ASSIGN M1,Mp is emitted and released for execution.

NOP M1,M2,M3,M4

Input cells: M2,M3,M4; Output cell: M1

M1 is defined to be ready and true when all of M2,M3,M4 are ready.

STEP M1,M2,M3,M4

Input cells: M1,M2,M3; Output cells: M1,M4

The content of M2 is added to the content of M1 and the result is left in M1. M3 is a dummy which triggers the step. M4 is set to the same value as M2 after the step is complete. M4 serves as a dummy variable to trigger other instructions.

TEST M1,M2,M3,M4

Input cells: M2,M3,M4; Output cell: M1

If M4 >= 0, this instruction behaves like <= M1,M2,M3.
If M4 < 0, this instruction behaves like >= M1,M2,M3.

APPENDIX C

IBM SYSTEM/360 MATRIX MULTIPLICATION PROGRAM

The following program, written in IBM System/360 Assembler Language (23)(25), multiplies together two square 2 X 2 matrices. The operand matrices are assumed to be stored in row-major order in the storage areas labelled A and B; the product matrix is left in row-major order in the storage area labelled C. To convert the program to multiply square N X N matrices, it is necessary only to increase the storage areas A, B, and C to accommodate N squared words each, and to change the constant N4 to contain 4 * N.

```
* REG. 0 WILL CONTAIN 4 * N
* REG. 1 WILL CONTAIN 4 * I
* REG. 2 WILL CONTAIN 4 * J
* REG. 3 WILL CONTAIN 4 * K
* REG. 12 WILL CONTAIN 4
*
A      DS   4F
B      DS   4F
C      DS   4F
N4     DC   F'8'
```

```
        LA    12,4        PUT 4 IN R12
        L     0,N4        PUT 4*N IN R0
        LR    1,12        SET I=1 (4*I=4)
ILOOP   LR    2,12        SET J=1 (4*J=4)
JLOOP   LR    5,1
        MR    4,0
        AR    5,2         R5 NOW CONTAINS DISPLACEMENT(I,J)
        SR    10,10       ZERO R10
        LR    3,12        SET K=1 (4*K=4)
KLOOP   LR    7,1
        MR    6,0
        AR    7,3         R7 NOW CONTAINS DISPLACEMENT(I,K)
        LR    9,3
        MR    8,0
        AR    9,2         R9 NOW CONTAINS DISPLACEMENT(K,J)
        LE    11,A(7)     LOAD A(I,K) INTO R11
        ME    11,B(9)     MULTIPLY A(I,K) * B(K,J)
        AER   10,11       ADD PRODUCT TO R10
        AR    3,12        INCREMENT K
        CR    3,0         IF K <= N,
        BC    12,KLOOP        GO TO KLOOP
        ST    10,C(5)     STORE R10 INTO C(I,J)
        AR    1,12        INCREMENT J
        CR    2,0         IF J <= N,
        BC    12,JLOOP        GO TO JLOOP
        AR    1,12        INCREMENT I
        CR    1,0         IF I <= N,
        BC    1,ILOOP         GO TO ILOOP
```

APPENDIX D

SAMPLE MATRIX MULTIPLICATION PROGRAM

The following SAMPLE program multiplies together two square matrices A and B and leaves the product matrix in C. The program is applicable without modification to square matrices of any size.

```
BEGIN
    L ← LAST A;
    I ← <0 TO L>;
    J ← <0 TO L>;
    K ← <0 TO L>;
    T ↓ 'I' OF L ↓ 'J' OF L ↓ 'K' OF L ←
        A ↓ 'I' ↓ 'J'  *  B ↓ 'K' ↓ 'J';
    C ↓ 'I' OF L ↓ 'J' OF L ← + T ↓ 'I' ↓ 'J';
END.
```

LIST OF REFERENCES

(1) Adams, D. A. A Computation Model With Data Flow Sequencing. Computer Science Department Report CS-117, Stanford University, Stanford, California (Dec. 1968).

(2) Anderson, D. W., Sparacio, F. J., and Tomasulo, R. M. The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling. IBM J. of R. & D., 11, 1 (Jan. 1967) 8-24.

(3) Anderson, J. P. Program Structures for Parallel Processing. Comm. ACM, 8, 12 (Dec. 1965) 786-788.

(4) Baer, J. L. Graph Models of Computation in Computer Systems. Report 68-46, Department of Engineering, UCLA, Los Angeles, Calif. (Oct. 1968).

(5) Barnes, G. H. , Brown, R. M., Kato, M., Kuck, D. J., Slotnick, D. L., and Stokes, R. A. The ILLIAC IV Computer. IEEE Trans. on Computers, C-17, 8 (Aug. 1968) 746-757.

(6) Bernstein, A. J. Analysis of Programs for Parallel Processing. IEEE Trans. on Electronic Computers, EC-15 (October 1966) 757-763.

(7) Bingham, H. W., Fisher, D. A., and Reigel, E. W. Automatic Detection of Parallelism in Computer Programs. Burroughs Corp. Technical Report TR-67-4 (Nov. 1967).

(8) Bredt, T. H. and McCluskey, E. J. A Model for Parallel Computer Systems. Technical Report No. 5, SEL Digital Systems Laboratory, Stanford University, Stanford, California (April 1970).

(9) Bredt, T. H. Analysis of Parallel Systems. Technical Report No. 7, SEL Digital Systems Laboratory, Stanford University, Stanford, California (August 1970).

(10) Bredt, T. H. A Survey of Models for Parallel Computing. Technical Report No. 8, SEL Digital Systems Laboratory, Stanford University, Stanford, California (August 1970).

(11) Constantine, L. L. Control of Sequence and Parallelism in Modular Programs. Proc. 1968 SJCC, 409-414.

(12) Control Data 7600 Computer System, Preliminary Reference Manual. Pub. No. 60258200, Control Data Corp., St. Paul, Minnesota (1968).

(13) Conway, M. E. A Multiprpcessor System Design. Proc. 1963 FJCC, 139-146.

(14) Crane, B. A., and Githens, J. A. Bulk Processing in Distributed Logic Memory. IEEE Trans. on Electronic Computers, EC-14, 2 (Apr. 1965) 186-196.

(15) Dennis, J. B., and Van Horn, E. C. Programming Semantics for Multiprogrammed Computations. Comm. ACM, 9,3 (March 1966) 143-155.

(16) Estrin, G., Bussell, B., Turn, R., and Bibb, J. Parallel Processing in a Restructurable Computer System. IEEE Trans. on Electronic Computers, EC-12, (1963) 747-755.

(17) Floyd, R. W. Bounded Context Syntactic Analysis. Comm. ACM, 7,2 (Feb. 1964) 62-66.

(18) Gosden, J. A. Explicit Parallel Processing Description and Control. Proc. 1966 FJCC.

(19) Habermann, A. N. Prevention of System Deadlocks. Comm. ACM, 12,7 (July 1969) 373-385.

(20) Hellerman, H. Parallel Processing of Algebraic Expressions. IEEE Trans. on Electronic Computers, EC-15 (Feb. 1966) 82-91.

(21) Holland, J. H. A Universal Computer Capable of Executing an Arbitrary Number of Sub-programs Simultaneously. Proc. 1959 EJCC, 108-113.

(22) IBM System/360 Component Descriptions--2314 Direct Access Storage Facility and 2844 Auxiliary Storage Control (IBM Publication No. A26-3599-2) pp.26-28.

(23) IBM System/360 Disk and Tape Operating Systems Assembler Language (IBM Publication No. C24-3414-5) (January 1968).

(24) IBM System/360 Operating System PL/I (F) Language Reference Manual (IBM Publication No. C28-8201-2) (October 1969).

(25) IBM System/360 Principles of Operation (IBM Publication No. A22-6821-2) (February 1966)

(26) Iverson, K. A Programming Language. (New York: John Wiley, 1962).

(27) Karp, R. M., and Miller, R. E. Properties of a Model for Parallel Computations: Determinacy, Termination, and Queueing. SIAM J. Appl. Math., 14 (Nov. 1966) 1390-1411.

(28) Martin, D. and Estrin, G. Models of Computational Systems--Cyclic to Acyclic Graph Transformations. IEEE Trans. on Electronic Computers, EC-16 (Feb. 1967).

(29) McKeeman, W. M. An Approach to Computer Language Design. PhD Thesis, Computer Science Department, Stanford University (Aug. 1966).

(30) McKeeman, W. M., Horning, J. J., and Wortman, D. B. A Compiler Generator. (Prentice-Hall, 1970).

(31) Opler, A. Procedure-Oriented Language Statements to Facilitate Parallel Processing. Comm. ACM, 8, 5 (May 1965) 306-307.

(32) Rodriguez, J. E. A Graph Model for Parallel Computations. PhD Thesis, MIT, Department of Electric Engineering, Cambridge, Massachusetts (Sept. 1967).

(33) Slotnick, D. L., Borck, W. C., and McReynolds, R. C. The SOLOMON Computer. Proc. 1962 FJCC, 97-107.

(34) Stone, H. S. One-Pass Compilation of Arithmetic Expressions for a Parallel Processor. Comm. ACM, 10,4 (April 1967) 220-223.

(35) Tesler, L. G., and Enea, H. J. A Language Design for Concurrent Processes. Proc. 1968 SJCC, 403-408.

(36) Wilkins, S. Representations of Process Parallelisms. Report No. 328, Department of Computer Science, University of Illinois, Urbana, Illinois. (June 1969).

(37) Wirth, N. A Note on 'Program Structures for Parallel Processing'. Comm. ACM, 9,5 (May 1966) 320-321.

Unclassified

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)	2a. REPORT SECURITY CLASSIFICATION
Digital Systems Laboratory	Unclassified
	2b. GROUP

3. REPORT TITLE

PARALLEL IMPLEMENTATION OF A SINGLE ASSIGNMENT LANGUAGE

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

Technical Report no. 13 January 1971

5. AUTHOR(S) *(First name, middle initial, last name)*

Donald D. Chamberlin

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
January 1971	178	37

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
N-00014-67-A-0112-0044 (JSEP) b. PROJECT NO. NGR-05-020-337 (NASA) 7101 (JSEP) c. 7111 (NASA) d.	SEL-71-007
	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. DISTRIBUTION STATEMENT

This document has been approved for public release and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Joint Services Electronics Program; U.S. Army, U.S. Navy, and U.S. Air Force

13. ABSTRACT

This thesis describes a high-level computer programming language, called SAMPLE, and a parallel processing system to implement the language. SAMPLE belongs to the class of single-assignment languages, which have the property that statements are not necessarily executed in their order of appearance in the program; rather, each statement is triggered by the readiness of the data on which it depends. Because of this property, single-assignment languages are well adapted for parallel processing.

Rules are given for compiling SAMPLE programs into machine-level instructions, and a machine organization is described to execute the resulting code. During execution of a program, many processors are active simultaneously, each with its own independent instruction stream. Expandability and graceful degradation are intrinsic properties of the system organization.

Some experiments are described which simulate the behavior of the proposed system and compare it with a conventional, single-processor system. It is concluded that the proposed system offers a speed advantage over a conventional system, at the expense of increased processor costs and memory requirements.

DD FORM 1 NOV 65 1473

Unclassified

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
parallel computers						
multiprocessing						
single-assignment language						
computer organization						